

Join the discussion @ [p2p.wrox.com](http://p2p.wrox.com)



Wrox Programmer to Programmer™



Beginning Visual C# 2010

# C# 入门经典

## (第5版)



(美) Karli Watson  
Christian Nagel  
齐立波  
黄 静

等著  
翻译  
审校

清华大学出版社



# 全面讲解C# 2010和.NET架构编程知识 为您编写卓越C# 2010程序奠定坚实基础

C#入门经典系列是屡获殊荣的C#名著和超级畅销书。最新版的《C#入门经典(第5版)》全面讲解C# 2010基础知识,浓墨重彩地描述Web和Windows编程以及数据访问(数据库和XML)等内容,详细介绍C#编程工具以及Visual Studio 2010中的Visual C# 2010开发环境。贯穿全书的分步说明和极富启迪意义的示例指引您使用高效C# 2010代码得心应手地编写程序。

## 本书内容

- ◆ 解释变量和表达式等基本C# 2010语法知识
- ◆ 介绍泛型的含义和用法
- ◆ 讨论Windows编程和Windows窗体
- ◆ 介绍C#改进内容、lambda表达式和扩展方法
- ◆ 解释Windows应用程序部署方法
- ◆ 讨论XML并简要介绍LINQ
- ◆ 深入探讨调试和错误处理方法
- ◆ 演示有效WPF和WCF技术

Karli Watson是Infusion Development 公司高级顾问,并担任Boost.net的技术架构师和IT自由撰稿人、作家和开发人员。他曾编著多本.NET(尤其是C#)书籍,极擅长以浅显易懂的方式阐明复杂技术主题。

Christian Nagel是微软技术代言人、微软MVP,拥有逾25年的软件开发经验。Christian熟悉各种语言和平台,曾编写多本.NET图书,并多次在国际会议上发表重要演讲。

**Wrox Beginning guides** are crafted to make learning programming languages and technologies easier than you think, providing a structured, tutorial format that will guide you through all the techniques involved.

## 源代码下载及技术支持

<http://www.wrox.com>

<http://www.tupwk.com.cn/downpage>

**Wrox**  
An Imprint of  
**WILEY**

上架建议: 编程语言/C#(.NET)  
读者信箱: [wkservice@vip.163.com](mailto:wkservice@vip.163.com)  
投稿信箱: [bookservice@263.net](mailto:bookservice@263.net)



# wrox.com

## Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

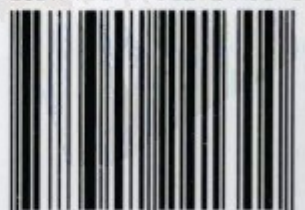
## Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

## Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

ISBN 978-7-302-24130-0



9 787302 241300 >

定价: 99.80元



# C#入门经典

(第5版)

(美)	Karli Watson	等著
	Christian Nagel	
	齐立波	翻译

清华大学出版社

北 京





Karli Watson, Christian Nagel, et al.

Beginning Visual C# 2010

EISBN: 978-0-470-50226-6

Copyright © 2010 by Wiley Publishing, Inc. Indianapolis, Indiana.

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2010-2530

本书封面贴有 Wiley 公司防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

#### 图书在版编目(CIP)数据

C#入门经典(第5版)/(美)沃森(Watson, K.), (美)内格尔(Nagel, C.) 等著; 齐立波 翻译; 黄静 审校.

—北京: 清华大学出版社, 2010.12

书名原文: Beginning Visual C# 2010

ISBN 978-7-302-24130-0

I. C… II. ①沃… ②内… ③齐… ④黄… III. C语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2010)第 210906 号

责任编辑: 王 军 韩宏志

装帧设计: 孔祥丰

责任校对: 胡雁翎

责任印制: 何 芊

出版发行: 清华大学出版社

地 址: 北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编: 100084

社 总 机: 010-62770175

邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 清华大学印刷厂

装 订 者: 三河市新茂装订有限公司

经 销: 全国新华书店

开 本: 185×260 印 张: 56.75 字 数: 1523 千字

版 次: 2010 年 12 月第 1 版 印 次: 2010 年 12 月第 1 次印刷

印 数: 1~5000

定 价: 99.80 元

产品编号: 035690-01



# 作者简介

Karli Watson 是 Infusion Development ([www.infusion.com](http://www.infusion.com)) 的顾问, Boost.net ([www.boost.net](http://www.boost.net)) 的技术架构师和 IT 自由撰稿专业人士、作家和开发人员。他主攻 .NET (尤其是 C# 和后来的 WPF), 为几家出版商编写了多本围绕这个领域的图书。他擅长以便于任何有学习热情的人理解的方式阐述复杂的理念, 并投入了大量时间研究新技术, 找出可教给其他人的新东西。

在工作之余(这种时间似乎很少), Karli 喜欢到山上滑雪, 或者尝试发表他的小说。他喜欢穿颜色鲜亮的衣服, 他的网址是 [www.twitter.com/karlequin](http://www.twitter.com/karlequin), 也许有一天他自己会建立一个网站。Karli 编写了本书的 1~14、12、25 和 26 章。

Christian Nagel 是 Microsoft 区域总监、Microsoft MVP, 是 Thinktecture 的合作伙伴, CN 创新技术的拥有者, 他是一位软件架构师和开发人员, 为开发 Microsoft .NET 解决方案提供培训和咨询服务。他拥有超过 25 年的软件开发经验。Christian 从 PDP 11 和 VAX/VMS 系统开始踏入其计算机生涯, 此后接触了各种语言 and 平台。自从 2000 年以来, (那时 .NET 还只是一个技术框架) 他就开始使用各种 .NET 技术建立大量的 .NET 解决方案。他具备深厚的 Microsoft 技术功底, 编写了大量 .NET 图书, 并获得了 Microsoft 认证培训师和专业开发人员的证书。Christian 在国际会议发表演讲, 例如 echEd 和 Tech Days, 并启动 INETA Europe 来支持 .NET 用户组。可以通过网站 [www.cninnovation.com](http://www.cninnovation.com) 和 [www.thinktecture.com](http://www.thinktecture.com) 联系 Christian, 在 [www.twitter.com/christiannagel](http://www.twitter.com/christiannagel) 上可以了解有关他的一些信息。Christian 编写了本书的 17~20 章。

Jacob Hammer Pedersen 是 Elbek & Vejrup 的一位资深应用程序开发人员, 他刚能拼写 BASIC 时就开始编程, BASIC 也是他使用的第一种编程语言。在 20 世纪 90 年代早期, 他开始使用 Pascal 在 PC 上编程, 不久就改用 C++, 目前, 他仍非常迷恋 C++。90 年代中期, 他的兴趣又改变了, 这次是 Visual Basic。2000 年夏, 他发现了 C#, 之后开始满心欢喜地研究这门语言。他主要工作在 Microsoft 平台上, 其他的工作领域包括 MS Office 开发、SQL Server、COM 和 Visual Basic.Net。

Jacob 是丹麦人, 工作生活在丹麦的奥尔胡斯市, 他编写了本书的 15、16 和 22 章。

Jon D. Reid 是 Metrix LLC 的一位软件工程经理, Metrix LLC 是 Microsoft 环境的区域服务管理软件的 ISV。他与他人合作编写了各种 .NET 图书, 包括 *Beginning Visual C# 2008*、*Beginning C# Databases: From Novice to Professional*、*Pro Visual Studio .NET* 等, Jon 编写了本书的第 23 和 24 章。

Morgan Skinner 在校时就学习 Sinclair ZX80, 开始了计算机生涯, 当时他对教师编写的一些代码不感兴趣, 便开始用汇编语言编程, 从那时开始他使用了所有的语言 and 平台, 包括 VAX 宏汇编、Pascal、Modula2、Smalltalk、X86 汇编语言、PowerBuilder、C/C++、VB 和目前的 C#, 自从 2000 年发布 PDC 以来, 他就用 .NET 编程, 而且非常喜欢 .NET, 所以在 2001 年加盟 Microsoft 公司, 他现在是开发人员的主要支持人员, 而且花费了大多数时间帮助客户使用 C#。Morgan 编写了本书的第 27 章。在 [www.morganskinner.com](http://www.morganskinner.com) 上可以联系到 Morgan。



# 技术编辑简介

Doug Holland 从 2007 年 3 月起担任英特尔公司的蓝带.NET 架构师和开发人员，是 Visual Computing Group 的成员，目前在高级工具和开发团队中工作，主要从事芯片集和驱动程序测试。Doug Holland 获得了牛津大学软件工程专业的硕士学位，已荣获 Microsoft MVP 和 Intel Black Belt Developer 奖。在工作之余，Doug 喜欢与妻子和 4 个孩子在一起享受快乐家庭生活，他还是 Civil Air Patrol/U.S. Air Force Auxiliary 的一位官员。除了构建和开发软件之外，Doug 还常常在加州的本地机场亲自驾驶 Cessnas 飞机在高空翱翔。





# 前言

C#是 Microsoft 在 2000 年 7 月推出 .NET Framework 的第 1 版时提供的一种全新语言。C# 迅速流行开来，成为使用 .NET Framework 的 Windows 和 Web 开发人员无可争议的选择。他们喜欢 C# 的一个原因是其派生于 C/C++ 的简洁明了的语法，这种语法简化了以前一些给程序员带来困扰的问题。尽管做了这些简化，但 C# 仍保持了 C++ 原有的功能，所以现在没有理由不从 C++ 转向 C#。C# 语言并不难，也非常适合于学习基本编程技术。易于学习，再加上 .NET Framework 的功能，使 C# 成为开始您编程生涯的绝佳方式。

C# 的最新版本 C# 4 是 .NET Framework 4 的一部分，它建立在已有的成功基础之上，还添加了一些更吸引人的功能。Visual Studio 的最新版本 Visual Studio 2010 和开发工具的 Express 系列(包括 Visual C# 2010 Express)也有许多变化和改进，这大大简化了编程工作，显著提高了效率。

本书将全面介绍 C# 编程的所有知识，从该语言本身一直到 Windows 和 Web 编程，再到数据源的使用，最后是一些新的高级技术。我们还将学习 Visual C# 2010 Express、Visual Web Developer 2010 Express 和 Visual Studio 2010 的功能和利用它进行应用程序开发的各种方式。

本书文笔优美流畅，阐述清晰，每一章都以前面章节的内容为基础，便于读者掌握高级技术。每个概念都会根据需要来介绍和讨论，而不会突然冒出某个技术术语来妨碍读者的阅读和理解。本书尽量减少使用的技术术语数量，但如果需要，将根据上下文进行正确的定义和布置。

本书的作者都是各自领域的专家，都是 C# 语言和 .NET Framework 的爱好者，没有人比他们更有资格讲授 C# 了，他们将在您掌握从基本规则到高级技术的过程中为您保驾护航。除了基础知识之外，本书还有许多有益的提示、练习、完全成熟的示例代码(可以从 [p2p.wrox.com](http://p2p.wrox.com) 上下载)，在您的职业生涯中一定会反复用到它们。

本书将毫无保留地传授这些知识，希望读者能通过阅读本书成长为最优秀的程序员。

## 0.1 本书读者对象

本书主要针对想学习如何使用 .NET Framework 编写 C# 程序的所有人。本书前面的章节介绍该语言本身，读者不需要具备任何编程经验。以前对其他语言有一定了解的开发人员，会觉得这些章节的内容非常熟悉。C# 语法的许多方面都与其他语言相同，许多结构对所有的编程语言来说都是相通的(例如，循环和分支结构)。但是，即使是有经验的程序员也可以从这些章节中获益，理解这些技术应用于 C# 的特征。

如果读者是编程新手，就应从头开始学习。如果读者对 .NET Framework 比较陌生，但知道如何编程，就应阅读第 1 章，然后快速跳读后面几章，这样就能掌握 C# 语言的应用方式了。如果读者知道如何编程，但以前从未接触过面向对象的编程语言，就应从第 8 章开始阅读以后的章节。



如果读者对 C#语言比较了解,就可以集中精力学习详细论述最新 .NET Framework 和 C#语言开发的章节,尤其是集合、泛型和 C# 4 语言的新增内容(第 11~14 章),或者完全跳过本书的第 I 部分,从第 15 章开始学习。

本书章节的编排方式可以达到两个目的:可以按顺序阅读这些章节,将其视为 C#语言的一个完整教程。还可以按照需要深入学习这些章节,将其作为一本参考资料。

除了核心内容之外,从第 3 章开始,每章末尾还包含一组练习,完成这些练习有助于读者理解所学的内容。练习包括简单的选择题、判断题以及需要修改或建立应用程序的较难问题。练习答案在 [www.wrox.com](http://www.wrox.com) 的本书 Web 页面上和 <http://www.tupwk.com.cn> 联机提供。

## 0.2 本版的新内容

本书特别注重与 C# 4、.NET 4 的一致性。每一章都进行了彻底的检查,删除了不太相关的内容,增加了新材料。所有代码都在最新版本的开发工具上进行了测试,所有屏幕图都在 Windows 7 上重新截取,以提供最新的窗口和对话框。

尽管我们不喜欢承认失误,但还是修订了前几版中的错误,处理了许多其他的读者评论。我们希望不要出现太多的新错误,但一旦发现了错误,我们的 Web 专家就会联机修改它们。

本版本的亮点包括:

- 增加并改进了代码示例。
- 涵盖 C# 4 的所有新内容,包括简单的语言改进,例如方法的命名参数和可选参数,还包括高级技术,例如泛型类型中的变体。
- 十分合理地介绍高级技术,重点是适合于新手、较易理解的内容。

## 0.3 本书结构

本书分为 6 个部分。

- 前言:概述本书的内容。
- C#语言:介绍了 C#语言的所有内容,从基础知识到面向对象的技术,一应俱全。
- Windows 编程:介绍如何用 C#编写 Windows 应用程序,如何部署它们。
- Web 编程:描述 Web 应用程序的开发、Web 服务和 Web 应用程序的部署。
- 数据访问:介绍如何在应用程序中使用数据,包括存储在硬盘文件上的数据、以 XML 格式存储的数据和数据库中的数据。
- 其他技术:讲述使用 C#和 .NET Framework 的一些额外方式,包括由 .NET 3.0 引入然后经 .NET 4 改进的 WPF、WCF 和 WF 技术。

下面介绍本书 5 个重要部分中的章节。

### 0.3.1 C#语言(第 1~14 章)

第 1 章介绍 C#及其与 .NET 的关系,了解在这个环境下编程的基础知识,以及 Visual C# 2010



Express(VCE)和 Visual Studio 2010(VS)与它的关系。

第 2 章开始介绍如何编写 C#应用程序,学习 C#的语法,并将 C#和样例命令行、Windows 应用程序结合起来使用。这些示例将说明 C#如何快速轻松地启动和运行,并附带介绍 VCE 和 VS 开发环境以及本书将要使用的基本窗口和工具。

第 3 章介绍 C#语言的更多基础知识,分析变量的含义以及如何操纵它们。第 4 章将用流程控制(循环和分支)改进应用程序的结构,第 5 章介绍一些高级变量类型,如数组。第 6 章开始以函数形式封装代码,这样就更易于执行重复的操作,使代码更容易让人理解。

从第 7 章开始将运用 C#语言的基础知识,调试应用程序。这包括在运行应用程序时输出跟踪信息,使用 VS 查找错误,在强大的调试环境中找出解决问题的办法。

第 8 章将学习面向对象编程(Object-Oriented Programming, OOP)。首先了解这个术语的含义,回答“什么是对象?”。OOP 初看起来是较难的问题。我们将用一整章的篇幅来介绍它,解释对象的强大之处。直到本章的最后才会使用 C#代码。

第 9 章将理论应用于实践,开始在 C#应用程序中使用 OOP 时,一切都会发生变化,而这才体现出 C#的真正威力。第 10 章首先介绍如何定义类和接口,然后探讨类成员(包括字段、属性和方法),在这一章的最后将开始创建一个扑克牌游戏应用程序,这个应用程序将在几章中开发完成,它非常有助于理解 OOP。

学习了 OOP 在 C#中的工作原理后,第 11 章将介绍几种常见的 OOP 场景,包括处理对象集合、比较和转换对象。第 12 章讨论 .NET 2.0 中 C#的一个非常有用的特性——泛型,利用它可以创建非常灵活的类。第 13 章通过一些其他技术和事件(它在 Windows 编程中非常重要)结束 C#语言和 OOP 的讨论。最后,第 14 章介绍 C# 3.0 和 4 中引入的新特性。

### 0.3.2 Windows 编程(第 15~17 章)

第 15 章开始介绍 Windows 编程的概念,理解在 VCE 和 VS 中如何实现 Windows 编程。这一章也是从基础知识开始介绍,并在本章和第 16 章中逐渐介绍较复杂的内容。第 16 章学习如何在应用程序中使用 .NET Framework 提供的各种控件。我们将简要论述 .NET 如何以图形化方式建立 Windows 应用程序,以最少的时间和精力创建高级应用程序。

第 17 章讨论应用程序的部署,包括建立安装程序,以使用户快速安装和运行应用程序。

### 0.3.3 Web 编程(第 18~20 章)

这个部分的结构与 Windows 编程部分类似。首先,第 18 章描述了构成最简单 Web 应用程序的控件,如何把它们组合在一起,让它们使用 ASP.NET 执行任务。接着介绍了更高级的技术、ASP.NET AJAX、各种控件、Web 上下文下的状态管理以及 Web 标准的遵循。

第 19 章将走入 Web 服务的精彩世界,它可以编程访问 Internet 上的信息和功能,可以把复杂数据和功能以独立于平台的方式嵌入 Web 和 Windows 应用程序。这一章讨论如何使用和创建 Web 服务,以及 .NET 提供的其他工具,如安全性。

最后,第 20 章探讨 Web 应用程序和服务的部署,尤其是可以通过单击按钮把应用程序发布到 Web 上的 VS 和 VWD 特性。

### 0.3.4 数据访问(第21~24章)

第21章介绍了应用程序如何将数据保存到磁盘以及如何检索磁盘上的数据(作为简单的文本文件或者更复杂的数据表示方式)。这一章还将讨论如何压缩数据,如何操纵旧数据(例如,用逗号分隔的值(CSV)文件),如何监视和处理文件系统的变化。

第22章学习数据交换的事实标准XML。之前的章节接触过XML几次,而这一章将了解XML的基本规则,论述XML的所有功能。

本部分其余章节介绍LINQ(这是内置于.NET Framework最新版本中的查询语言)。第23章简要介绍LINQ,第24章使用LINQ访问数据库和其他数据。

### 0.3.5 其他技术(第25~27章)

本书最后一部分将讨论.NET Framework最新版本中出现的几项新技术。第25章介绍Windows Presentation Foundation(WPF),了解它给Windows和Web开发带来哪些重大的变化。第26章介绍Windows Communication Foundation(WCF),它把Web服务的概念扩展和改进为一种企业级通信技术。本书最后一章是第27章,介绍了Windows Workflow Foundation(WF),它允许在应用程序中执行 workflow 功能,因此可以定义一些操作,这些操作由外部的交互操作控制,按特定顺序执行,这对许多类型的应用程序都很有帮助。

## 0.4 使用本书的要求

本书中C#和.NET Framework的代码和描述都适用于.NET 4。除了Framework之外,不需要其他东西就可以理解本书的这个方面,但许多示例都需要使用开发工具。本书将Visual C# 2010 Express作为主要开发工具,一些章节则使用了Visual Web Developer 2010 Express。另外,一些功能只能在Visual Studio 2010中使用,这会在相应的地方明确指出。

## 0.5 源代码

在读者学习本书中的示例时,可以手工输入所有代码,也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点<http://www.wrox.com/>或[www.tupwk.com.cn/downpage](http://www.tupwk.com.cn/downpage)上下载。登录到站点<http://www.wrox.com/>,使用Search工具或使用书名列表就可以找到本书。接着单击本书细目页面上的Download Code链接,就可以获得所有源代码。

注释:

由于许多图书的标题都很类似,所以按ISBN搜索是最简单的,本书英文版的ISBN是978-0-470-50226-6。

在下载了代码后,只需用自己喜欢的解压缩软件对它进行解压缩即可。另外,也可以进入<http://www.wrox.com/dynamic/books/download.aspx>上的Wrox代码下载主页,查看本书和其他Wrox图书的所有代码。



## 0.6 勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果您在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

请给 [wkservice@vip.163.com](mailto:wkservice@vip.163.com) 发电子邮件，我们就会检查您的反馈信息，如果是正确的，我们将在本书的后续版本中采用。在本书编辑过程中，我们接受了热心读者白爽针对第4版中文译著提出的一些修改意见，在此特向白爽表示衷心感谢。

要在网站上找到本书英文版的勘误表，可以登录 <http://www.wrox.com>，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 [www.wrox.com/misc-pages/booklist.shtml](http://www.wrox.com/misc-pages/booklist.shtml)。

## 0.7 P2P.WROX.COM

要与作者和同行讨论，请加入 [p2p.wrox.com](http://p2p.wrox.com) 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给您传送感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 <http://p2p.wrox.com> 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

- (1) 进入 [p2p.wrox.com](http://p2p.wrox.com)，单击 Register 链接。
- (2) 阅读使用协议，并单击 Agree 按钮。
- (3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击 Submit 按钮。
- (4) 您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

注释：

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要想让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

# 目 录

第 I 部分 C#语言	
第 1 章 C#简介 .....	3
1.1 .NET Framework 的含义 .....	3
1.1.1 .NET Framework 的内容 .....	4
1.1.2 使用.NET Framework 编写 应用程序 .....	4
1.2 C#的含义 .....	7
1.2.1 用 C#能编写什么样的 应用程序 .....	7
1.2.2 本书中的 C# .....	8
1.3 Visual Studio 2010 .....	8
1.3.1 Visual Studio 2010 Express 产品 .....	9
1.3.2 解决方案 .....	9
1.4 小结 .....	9
1.5 本章要点 .....	10
第 2 章 编写C#程序 .....	11
2.1 开发环境 .....	12
2.1.1 Visual Studio 2010 .....	12
2.1.2 Visual C# 2010 Express Edition .....	14
2.2 控制台应用程序 .....	15
2.2.1 Solution Explorer .....	19
2.2.2 Properties 窗口 .....	20
2.2.3 Error List 窗口 .....	20
2.3 Windows Forms 应用程序 .....	21
2.4 小结 .....	25
2.5 本章要点 .....	25
第 3 章 变量和表达式 .....	27
3.1 C#的基本语法 .....	27
3.2 C#控制台应用程序的 基本结构 .....	30
3.3 变量 .....	31
3.3.1 简单类型 .....	31
3.3.2 变量的命名 .....	35
3.3.3 字面值 .....	36
3.3.4 变量的声明和赋值 .....	38
3.4 表达式 .....	39
3.4.1 数学运算符 .....	39
3.4.2 赋值运算符 .....	43
3.4.3 运算符的优先级 .....	44
3.4.4 名称空间 .....	45
3.5 小结 .....	47
3.6 练习 .....	48
3.7 本章要点 .....	49
第 4 章 流程控制 .....	51
4.1 布尔逻辑 .....	51
4.1.1 布尔赋值运算符 .....	54
4.1.2 按位运算符 .....	55
4.1.3 运算符优先级的更新 .....	59
4.2 goto 语句 .....	60
4.3 分支 .....	61
4.3.1 三元运算符 .....	61
4.3.2 if 语句 .....	61
4.3.3 switch 语句 .....	65
4.4 循环 .....	68
4.4.1 do 循环 .....	68
4.4.2 while 循环 .....	71
4.4.3 for 循环 .....	73
4.4.4 循环的中断 .....	77
4.4.5 无限循环 .....	78



4.5	小结	78	7.2.1	try...catch...finally	153
4.6	练习	79	7.2.2	列出和配置异常	157
4.7	本章要点	79	7.2.3	异常处理的注意事项	158
<b>第5章</b>	<b>变量的更多内容</b>	<b>81</b>	7.3	小结	159
5.1	类型转换	81	7.4	练习	159
5.1.1	隐式转换	82	7.5	本章要点	159
5.1.2	显式转换	83	<b>第8章</b>	<b>面向对象编程简介</b>	<b>161</b>
5.1.3	使用 Convert 命令进行 显式转换	86	8.1	面向对象编程的含义	162
5.2	复杂的变量类型	89	8.1.1	对象的含义	162
5.2.1	枚举	89	8.1.2	一切皆对象	165
5.2.2	结构	93	8.1.3	对象的生命周期	165
5.2.3	数组	96	8.1.4	静态和实例类成员	166
5.3	字符串的处理	102	8.2	OOP 技术	167
5.4	小结	106	8.2.1	接口	167
5.5	练习	107	8.2.2	继承	169
5.6	本章要点	108	8.2.3	多态性	171
<b>第6章</b>	<b>函数</b>	<b>109</b>	8.2.4	对象之间的关系	172
6.1	定义和使用函数	110	8.2.5	运算符重载	173
6.1.1	返回值	111	8.2.6	事件	174
6.1.2	参数	113	8.2.7	引用类型和值类型	174
6.2	变量的作用域	120	8.3	Windows 应用程序中的 OOP	175
6.2.1	其他结构中变量的作用域	122	8.4	小结	177
6.2.2	参数和返回值与全局数据	124	8.5	练习	177
6.3	Main()函数	125	8.6	本章要点	178
6.4	结构函数	128	<b>第9章</b>	<b>定义类</b>	<b>179</b>
6.5	函数的重载	128	9.1	C#中的类定义	179
6.6	委托	130	9.2	System.Object	184
6.7	小结	133	9.3	构造函数和析构函数	185
6.8	练习	133	9.4	VS 和 VCE 中的 OOP 工具	190
6.9	本章要点	134	9.4.1	Class View 窗口	190
<b>第7章</b>	<b>调试和错误处理</b>	<b>135</b>	9.4.2	对象浏览器	192
7.1	VS 和 VCE 中的调试	135	9.4.3	添加类	193
7.1.1	非中断(正常)模式下的 调试	136	9.4.4	类图	194
7.1.2	中断模式下的调试	144	9.5	类库项目	196
7.2	错误处理	152	9.6	接口和抽象类	199
			9.7	结构类型	201
			9.8	浅度和深度复制	203
			9.9	小结	203

9.10	练习	204	11.2	比较	263
9.11	本章要点	204	11.2.1	类型比较	263
第 10 章	定义类成员	205	11.2.2	值比较	268
10.1	成员定义	205	11.3	转换	283
10.1.1	定义字段	206	11.3.1	重载转换运算符	284
10.1.2	定义方法	206	11.3.2	as 运算符	285
10.1.3	定义属性	207	11.4	小结	286
10.1.4	在类图中添加成员	212	11.5	练习	286
10.1.5	重构成员	215	11.6	本章要点	287
10.1.6	自动属性	216	第 12 章	泛型	289
10.2	类成员的其他议题	217	12.1	泛型的概念	289
10.2.1	隐藏基类方法	217	12.2	使用泛型	291
10.2.2	调用重写或隐藏的 基类方法	219	12.2.1	可空类型	291
10.2.3	嵌套的类型定义	220	12.2.2	System.Collections.Generic 名称空间	297
10.3	接口的实现	220	12.3	定义泛型类型	307
10.4	部分类定义	224	12.3.1	定义泛型类	308
10.5	部分方法定义	225	12.3.2	定义泛型接口	319
10.6	示例应用程序	227	12.3.3	定义泛型方法	319
10.6.1	规划应用程序	227	12.3.4	定义泛型委托	321
10.6.2	编写类库	228	12.4	变体	321
10.6.3	类库的客户应用程序	235	12.4.1	协变	322
10.7	Call Hierarchy 窗口	236	12.4.2	抗变	323
10.8	小结	237	12.5	小结	324
10.9	练习	237	12.6	练习	324
10.10	本章要点	238	12.7	本章要点	325
第 11 章	集合、比较和转换	239	第 13 章	其他 OOP 技术	327
11.1	集合	239	13.1	::运算符和全局名称空间 限定符	327
11.1.1	使用集合	240	13.2	定制异常	329
11.1.2	定义集合	246	13.3	事件	331
11.1.3	索引符	247	13.3.1	事件的含义	331
11.1.4	给 CardLib 添加 Cards 集合	250	13.3.2	处理事件	332
11.1.5	关键字值集合和 IDictionary	252	13.3.3	定义事件	334
11.1.6	迭代器	254	13.4	扩展和使用 CardLib	343
11.1.7	深复制	259	13.5	小结	351
11.1.8	给 CardLib 添加深复制	261	13.6	练习	352
			13.7	本章要点	352



<b>第 14 章 C#语言的改进</b>	<b>353</b>
14.1 初始化器	353
14.1.1 对象初始化器	354
14.1.2 集合初始化器	356
14.2 类型推理	359
14.3 匿名类型	360
14.4 动态查找	364
14.4.1 dynamic 类型	365
14.4.2 IdynamicMetaObject- Provider	369
14.5 高级方法参数	369
14.5.1 可选参数	369
14.5.2 命名参数	371
14.5.3 命名参数和可选参数 的规则	375
14.6 扩展方法	375
14.7 Lambda 表达式	379
14.7.1 复习匿名方法	379
14.7.2 把 Lambda 表达式用于 匿名方法	380
14.7.3 Lambda 表达式的参数	383
14.7.4 Lambda 表达式的 语句体	384
14.7.5 Lambda 表达式用作委托 和表达式树	385
14.7.6 Lambda 表达式和集合	386
14.8 小结	388
14.9 练习	389
14.10 本章要点	390

## 第 II 部分 Windows 编程

<b>第 15 章 Windows 编程基础</b>	<b>393</b>
15.1 控件	393
15.1.1 属性	394
15.1.2 控件的定位、停靠和 对齐	395
15.1.3 Anchor 和 Dock 属性	395
15.1.4 事件	396
15.2 Button 控件	398

15.2.1 Button 控件的属性	398
15.2.2 Button 控件的事件	398
15.2.3 添加事件处理程序	399
15.3 Label 和 LinkLabel 控件	400
15.4 TextBox 控件	401
15.4.1 TextBox 控件的属性	401
15.4.2 TextBox 控件的事件	402
15.4.3 添加事件处理程序	404
15.5 RadioButton 和 CheckBox 控件	407
15.5.1 RadioButton 控件的 属性	408
15.5.2 RadioButton 控件的 事件	408
15.5.3 CheckBox 控件的属性	408
15.5.4 CheckBox 控件的事件	409
15.5.5 GroupBox 控件	409
15.6 RichTextBox 控件	412
15.6.1 RichTextBox 控件的 属性	412
15.6.2 RichTextBox 控件的 事件	413
15.7 ListBox 和 CheckedListBox 控件	418
15.7.1 ListBox 控件的属性	418
15.7.2 ListBox 控件的方法	419
15.7.3 ListBox 控件的事件	420
15.8 ListView 控件	422
15.8.1 ListView 控件的属性	422
15.8.2 ListView 控件的方法	424
15.8.3 ListView 控件的事件	424
15.8.4 ListViewItem	425
15.8.5 ColumnHeader	425
15.8.6 ImageList 控件	425
15.9 TabControl 控件	431
15.9.1 TabControl 控件的属性	432
15.9.2 使用 TabControl 控件	432
15.10 小结	434
15.11 练习	434

15.12 本章要点 .....	434	17.5 为 MDI Editor 创建安装 软件包 .....	480
<b>第 16 章 Windows 窗体的高级功能</b> ...	435	17.5.1 规划安装内容 .....	480
16.1 菜单和工具栏 .....	435	17.5.2 创建项目 .....	481
16.1.1 两个实质一样的控件 .....	436	17.5.3 项目属性 .....	482
16.1.2 使用 MenuStrip 控件 .....	436	17.5.4 安装编辑器 .....	485
16.1.3 手工创建菜单 .....	436	17.5.5 File System 编辑器 .....	485
16.1.4 ToolStripMenuItem 控件的其他属性 .....	438	17.5.6 File Types 编辑器 .....	488
16.1.5 给菜单添加功能 .....	438	17.5.7 Launch Condition 编辑器 .....	489
16.2 工具栏 .....	440	17.5.8 User Interface 编辑器 .....	490
16.2.1 ToolStrip 控件的属性 .....	441	17.6 生成项目 .....	493
16.2.2 ToolStrip 的项 .....	441	17.7 安装 .....	493
16.2.3 StatusStrip 控件 .....	445	17.7.1 Welcome .....	494
16.2.4 StatusStripStatusLabel 的属性 .....	446	17.7.2 Read Me .....	494
16.3 SDI 和 MDI 应用程序 .....	448	17.7.3 License Agreement .....	495
16.4 生成 MDI 应用程序 .....	449	17.7.4 Optional Files .....	495
16.5 创建控件 .....	456	17.7.5 选择安装文件夹 .....	496
16.5.1 调试用户控件 .....	461	17.7.6 确认安装 .....	496
16.5.2 扩展 LabelTextbox 控件 .....	461	17.7.7 进度 .....	497
16.6 小结 .....	464	17.7.8 完成安装 .....	497
16.7 练习 .....	464	17.7.9 运行应用程序 .....	498
16.8 本章要点 .....	464	17.7.10 卸载 .....	498
<b>第 17 章 部署 Windows 应用程序</b> ...	465	17.8 小结 .....	498
17.1 部署概述 .....	465	17.9 练习 .....	499
17.2 ClickOnce 部署 .....	466	17.10 本章要点 .....	499
17.2.1 创建 ClickOnce 部署 .....	466		
17.2.2 用 ClickOnce 安装 应用程序 .....	474	<b>第III部分 Web 编程</b>	
17.2.3 创建和使用应用程序 的更新包 .....	476	<b>第 18 章 ASP.NET Web 编程</b> .....	503
17.3 Visual Studio 安装和部署 项目类型 .....	477	18.1 Web 应用程序概述 .....	503
17.4 Microsoft Windows 安装 程序结构 .....	478	18.2 ASP.NET 运行库 .....	504
17.4.1 Windows 安装程序术语 .....	478	18.3 创建简单的 Web 页面 .....	504
17.4.2 Windows 安装程序 的优点 .....	480	18.4 服务器控件 .....	512
		18.5 ASP.NET 回送 .....	513
		18.6 ASP.NET AJAX 回送 .....	518
		18.7 输入的有效性验证 .....	521
		18.8 状态管理 .....	525
		18.8.1 客户端的状态管理 .....	525
		18.8.2 服务器端的状态管理 .....	527



18.9 样式 .....	530	20.2 IIS 配置 .....	582
18.10 母版页 .....	535	20.3 复制 Web 站点 .....	584
18.11 站点导航 .....	540	20.4 发布 Web 站点 .....	587
18.12 身份验证和授权 .....	542	20.5 Windows 安装程序 .....	589
18.12.1 身份验证的配置 .....	543	20.5.1 创建安装程序 .....	589
18.12.2 使用安全控件 .....	546	20.5.2 安装 Web 应用程序 .....	591
18.13 读写 SQL Server 数据库 .....	549	20.6 小结 .....	592
18.14 小结 .....	556	20.7 练习 .....	593
18.15 练习 .....	556	20.8 本章要点 .....	593
18.16 本章要点 .....	556		
<b>第 19 章 Web 服务 .....</b>	<b>557</b>	<b>第 IV 部分 数据访问</b>	
19.1 使用 Web 服务的场合 .....	557	<b>第 21 章 文件系统数据 .....</b>	<b>597</b>
19.1.1 宾馆旅行社代理 应用程序 .....	558	21.1 流 .....	597
19.1.2 图书发布应用程序 .....	558	21.2 用于输入和输出的类 .....	598
19.1.3 客户应用程序的类型 .....	558	21.2.1 File 类和 Directory 类 .....	599
19.2 应用程序的体系结构 .....	558	21.2.2 FileInfo 类 .....	600
19.3 Web 服务的体系结构 .....	559	21.2.3 DirectoryInfo 类 .....	602
19.3.1 调用方法和 WSDL .....	559	21.2.4 路径名和相对路径 .....	602
19.3.2 调用方法 .....	560	21.2.5 FileStream 对象 .....	602
19.3.3 WS-I 规范 .....	561	21.2.6 StreamWriter 对象 .....	608
19.4 Web 服务和 .NET Framework .....	561	21.2.7 StreamReader 对象 .....	611
19.4.1 创建 Web 服务 .....	562	21.2.8 读写压缩文件 .....	617
19.4.2 客户程序 .....	563	21.3 序列化对象 .....	620
19.5 创建简单的 ASP.NET Web 服务 .....	564	21.4 监控文件系统 .....	625
19.6 测试 Web 服务 .....	567	21.5 小结 .....	631
19.7 实现 Windows 客户程序 .....	568	21.6 练习 .....	632
19.8 异步调用服务 .....	572	21.7 本章要点 .....	632
19.9 实现 ASP.NET 客户程序 .....	575	<b>第 22 章 XML .....</b>	<b>633</b>
19.10 传送数据 .....	576	22.1 XML 文档 .....	634
19.11 小结 .....	579	22.1.1 XML 元素 .....	634
19.12 练习 .....	580	22.1.2 特性 .....	635
19.13 本章要点 .....	580	22.1.3 XML 声明 .....	635
<b>第 20 章 部署 Web 应用程序 .....</b>	<b>581</b>	22.1.4 XML 文档的结构 .....	636
20.1 Internet Information Services .....	581	22.1.5 XML 名称空间 .....	636
		22.1.6 格式良好并有效的 XML .....	637
		22.1.7 验证 XML 文档 .....	638
		22.2 在应用程序中使用 XML .....	641

22.2.1 XML 文档对象模型.....	641	23.19 Join 查询.....	691
22.2.2 选择节点.....	650	23.20 小结.....	693
22.2.3 XPath.....	651	23.21 练习.....	693
22.3 小结.....	654	23.22 本章要点.....	693
22.4 练习.....	655	第 24 章 应用 LINQ.....	695
22.5 本章要点.....	655	24.1 LINQ 的变体.....	695
第 23 章 LINQ 简介.....	657	24.2 给数据库使用 LINQ.....	696
23.1 第一个 LINQ 查询.....	658	24.3 安装 SQL Server 和 Northwind 示例数据.....	696
23.1.1 用 var 关键字声明 结果变量.....	659	24.3.1 安装 SQL Server Express 2008.....	697
23.1.2 指定数据源: from 子句.....	660	24.3.2 安装 Northwind 示例 数据库.....	697
23.1.3 指定条件: where 子句.....	660	24.4 第一个 LINQ 数据库查询.....	697
23.1.4 指定元素: select 子句.....	660	24.5 浏览数据库关系.....	701
23.1.5 完成: 使用 foreach 循环.....	661	24.6 使用 LINQ to XML.....	703
23.1.6 延迟执行的查询.....	661	24.7 LINQ to XML 函数构造 方法.....	703
23.2 使用 LINQ 方法语法.....	661	24.8 保存和加载 XML 文档.....	707
23.2.1 LINQ 扩展方法.....	661	24.8.1 从字符串中加载 XML.....	710
23.2.2 查询语法和方法语法.....	662	24.8.2 已保存的 XML 文档 内容.....	710
23.3 排序查询结果.....	663	24.9 处理 XML 片段.....	710
23.4 orderby 子句.....	665	24.10 从数据库中生成 XML.....	713
23.5 用方法语法排序.....	665	24.11 查询 XML 文档的方法.....	715
23.6 查询大型数据集.....	667	24.12 使用 LINQ to XML 查询 成员.....	716
23.7 聚合运算符.....	669	24.12.1 Elements().....	717
23.8 查询复杂的对象.....	672	24.12.2 Descendants().....	717
23.9 投影: 在查询中创建新 对象.....	676	24.12.3 Attributes().....	719
23.10 投影: 方法语法.....	678	24.13 小结.....	721
23.11 单值选择查询.....	678	24.14 练习.....	721
23.12 Any 和 All.....	679	24.15 本章要点.....	722
23.13 多级排序.....	681	第 V 部分 其他技术	
23.14 多级排序方法语法: ThenBy.....	683	第 25 章 Windows Presentation Foundation.....	725
23.15 组合查询.....	683	25.1 WPF 的概念.....	726
23.16 Take 和 Skip.....	685		
23.17 First 和 FirstOrDefault.....	687		
23.18 集运算符.....	688		

25.1.1	WPF 给设计人员带来的好处	726
25.1.2	WPF 给 C#开发人员带来的好处	728
25.2	基本 WPF 应用程序的组成	729
25.3	WPF 基础	739
25.3.1	XAML 语法	740
25.3.2	桌面和 Web 应用程序	742
25.3.3	Application 对象	742
25.3.4	控件基础	743
25.3.5	控件的布局	751
25.3.6	控件的样式	760
25.3.7	触发器	764
25.3.8	动画	765
25.3.9	静态和动态资源	768
25.4	用 WPF 编程	773
25.4.1	WPF 用户控件	774
25.4.2	实现依赖属性	774
25.5	小结	784
25.6	练习	785
25.7	本章要点	785

## 第 26 章 Windows Communication

	Foundation	787
26.1	WCF 的含义	788
26.2	WCF 概念	788
26.2.1	WCF 通信协议	789
26.2.2	地址、端点和绑定	789
26.2.3	合同	791
26.2.4	消息模式	791
26.2.5	行为	792
26.2.6	驻留	792
26.3	WCF 编程	792
26.3.1	WCF 测试客户程序	800
26.3.2	定义 WCF 服务合同	802
26.3.3	自驻留的 WCF 服务	810
26.4	小结	816
26.5	练习	817
26.6	本章要点	817

## 第 27 章 Windows Workflow

	Foundation	819
27.1	Hello World	819
27.2	工作流和活动	821
27.2.1	If 活动	821
27.2.2	While 活动	822
27.2.3	Sequence 活动	822
27.3	变元和变量	823
27.4	定制活动	828
27.4.1	工作流扩展	830
27.4.2	活动的有效性验证	835
27.4.3	活动设计器	836
27.5	小结	838
27.6	练习	838
27.7	本章要点	838

附录 A	习题答案	839
------	------	-----



# 第 I 部分

## C# 语言

---

- 第 1 章 C#简介
- 第 2 章 编写 C#程序
- 第 3 章 变量和表达式
- 第 4 章 流程控制
- 第 5 章 变量的更多内容
- 第 6 章 函数
- 第 7 章 调试和错误处理
- 第 8 章 面向对象编程简介
- 第 9 章 定义类
- 第 10 章 定义类成员
- 第 11 章 集合、比较和转换
- 第 12 章 泛型
- 第 13 章 其他 OOP 技术
- 第 14 章 C#语言的改进



# 第 1 章

## C# 简介

### 本章内容:

---

- .NET Framework 的功能及其包含的内容
- .NET 应用程序的工作原理
- C#的概念及其与.NET Framework 的关系
- 用 C#创建.NET 应用程序的工具

本书第 I 部分将介绍使用 C# 语言所需的基础知识。第 1 章将概述 C#和.NET Framework, 包括这两项技术的含义、作用及相互关系。

首先讨论.NET Framework。这种技术包含的许多概念初看起来都不是很容易掌握的。也就是说, 我们必须在很短的篇幅里介绍许多新概念, 但是, 快速浏览这些基础知识对于理解如何利用 C#进行编程是非常重要的, 本书后面将详细论述这里提到的许多论题。

之后, 本章将讨论 C#本身, 包括它的起源以及与C++的类似之处。最后介绍本书使用的主要工具: Visual Studio 2010 (VS)和 Visual C# 2010 Express(VCE)。

### 1.1 .NET Framework 的含义

.NET Framework(现在是版本 4)是 Microsoft 为开发应用程序而创建的一个具有革命意义的平台。这句话最有趣的地方在于它的广义性, 但这是有原因的。首先, 注意这句话没有说“在 Windows 操作系统上开发应用程序”。尽管.NET Framework 的 Microsoft 版本运行在 Windows 操作系统上, 但以后将推出运行在其他操作系统上的版本, 例如 Mono, 它是.NET Framework 的开源版本(包含 C#编译器), 该版本可以运行在几个操作系统上, 包括各种 Linux 版本和 Mac OS。另外, 还可以在个人数字助手(PDA)类设备和一些智能电话上使用 Microsoft .NET Compact Framework(基本上是完整 .NET Framework 的一个子集)。使用.NET Framework 的一个重要原因是它可以作为集成各种操作系统的方式。

另外, 上面给出的.NET Framework 定义并未限制应用程序的类型。这是因为本来就没有限制。可以使用.NET Framework 创建 Windows 应用程序、Web 应用程序、Web 服务和其他各种类型的应



用程序。另外注意,对于 Web 应用程序,按照定义,它们是多平台的应用程序,因为任何带 Web 浏览器的系统都可以访问它们。最近新增了 Silverlight,这种类别还包含运行在客户浏览器内部的应用程序,以及仅以 HTML 格式显示 Web 内容的应用程序。

.NET Framework 的设计方式确保它可以用于各种语言,包括本书介绍的 C# 语言,以及 C++、Visual Basic、JScript,甚至一些旧的语言,如 COBOL。为此,还推出了这些语言的 .NET 版本,目前还在不断推出更多版本。所有这些语言都可以访问 .NET Framework,它们彼此之间还可以通信。C# 开发人员可以使用 Visual Basic 程序员编写的代码,反之亦然。

所有这些提供了意想不到的多样性,这也是 .NET Framework 具有诱人前景的部分原因。

### 1.1.1 .NET Framework 的内容

.NET Framework 主要包含一个非常大的代码库,可以在客户语言(如 C#)中通过面向对象编程技术(OOP)来使用这些代码。这个库分为多个不同的模块,这样就可以根据希望得到的结果来选择使用其中的各个部分。例如,一个模块包含 Windows 应用程序的构件,另一个模块包含网络编程的代码块,还有一个模块包含 Web 开发的代码块。一些模块还分为更具体的子模块,例如,在 Web 开发模块中,有用于建立 Web 服务的子模块。

其目的是,不同操作系统可以根据自己的特性,支持其中的部分或全部模块。例如,PDA 支持所有的核心 .NET 功能,但不需要某些更高级的模块。

部分 .NET Framework 库定义了一些基本类型。类型是数据的一种表达方式,指定其中最基础的部分(如 32 位带符号的整数),以便使用 .NET Framework 在各种语言之间进行交互操作。这称为通用类型系统(Common Type System, CTS)。

除了提供这个库以外,.NET Framework 还包含 .NET 公共语言运行库(Common Language Runtime, CLR),它负责管理用 .NET 库开发的所有应用程序的执行。

### 1.1.2 使用 .NET Framework 编写应用程序

使用 .NET Framework 编写应用程序,就是使用 .NET 代码库编写代码(使用支持 Framework 的任何一种语言)。本书用 VS 和 VCE 进行开发,VS 是一种强大的集成开发环境,支持 C#(以及托管和非托管 C++、Visual Basic 和其他一些语言)。VCE 是 VS 的一个删节版本(免费),仅支持 C#。这些环境的优点是便于把 .NET 功能集成到代码中。我们创建的代码完全是 C# 代码,但使用了 .NET Framework,并在需要时利用了 VS 和 VCE 中的其他工具。

为了执行 C# 代码,必须把它们转换为目标操作系统能够理解的语言,即本机代码(native code)。这种转换称为编译代码,由编译器执行。但在 .NET Framework 下,此过程包括两个阶段。

#### 1. CIL 和 JIT

在编译使用 .NET Framework 库的代码时,不是立即创建专用于操作系统的本机代码,而是把代码编译为通用中间语言(Common Intermediate Language, CIL)代码,这些代码并非专门用于任何一种操作系统,也非专门用于 C#。其他 .NET 语言,如 Visual Basic .NET 也可以在第一阶段编译为这种语言,开发 C# 应用程序时,这个编译步骤由 VS 或 VCE 完成。

显然,要执行应用程序,必须完成更多工作,这是 Just-In-Time(JIT)编译器的任务,它把 CIL 编译为专用于 OS 和目标机器结构的本机代码。这样 OS 才能执行应用程序。这里编译器的名称

Just-In-Time 反映了 CIL 代码仅在需要时才编译的事实。

过去,常常需要把代码编译为几个应用程序,每个应用程序都用于特定的操作系统和 CPU 结构。这通常是一种优化形式(例如,为了让代码在 AMD 芯片组上运行得更快),而且有时是非常重要的(例如,对于工作在 Win9x 和 WinNT/2000 环境下的应用程序)。现在就不必要了,因为顾名思义, JIT 编译器使用 CIL 代码,而 CIL 代码是独立于计算机、操作系统和 CPU 的。目前有几种 JIT 编译器,每种编译器都用于不同的结构,我们总能找到一个合适的编译器创建所需的本机代码。

这样,用户需要做的工作就比较少了。实际上,可以忽略与系统相关的细节,把注意力集中在代码的功能上就够了。



读者可以遇到过 Microsoft Intermediate Language(MSIL)或 IL, MSIL 是 CIL 原来的名称,许多开发人员仍沿用这个术语。

## 2. 程序集

在编译应用程序时,所创建的 CIL 代码存储在一个程序集中。程序集包括可执行的应用程序文件(这些文件可以直接在 Windows 上运行,不需要其他程序,其扩展名是.exe)和其他应用程序使用的库(其扩展名是.dll)。

除了包含 CIL 外,程序集还包含元信息(即程序集中包含的数据的信息,也称为元数据)和可选的资源(CIL 使用的其他数据,例如,声音文件和图片)。元信息允许程序集是完全自描述的。不需要其他信息就可以使用程序集,也就是说,我们不会遇到下述情形:不能把需要的数据添加到系统注册表中,而这种情形在使用其他平台进行开发时常常出现。

因此,部署应用程序就非常简单了,只需把文件复制到远程计算机上的目录下即可。因为不需要目标系统上的其他信息,所以只需从该目录中运行可执行文件即可(假定安装了.NET CLR)。

当然,不必把运行应用程序需要的所有信息都安装到一个地方。可以编写一些代码来执行多个应用程序所要求的任务。此时,通常把这些可重用的代码放在所有应用程序都可以访问的地方。在.NET Framework 中,这个地方是全局程序集缓存(Global Assembly Cache, GAC),把代码放在这个缓存中是很简单的,只需把包含代码的程序集放在包含该缓存的目录中即可。

## 3. 托管代码

在将代码编译为 CIL,再用 JIT 编译器将它编译为本机代码后,CLR 的任务还没有全部完成,还需要管理正在执行的用.NET Framework 编写的代码(这个执行代码的阶段通常称为运行时(runtime))。即 CLR 管理着应用程序,其方式是管理内存、处理安全性以及允许进行跨语言调试等。相反,不受 CLR 控制运行的应用程序属于非托管类型,某些语言如 C++可以用于编写这类应用程序,例如,访问操作系统的低级功能。但是在 C#中,只能编写在托管环境下运行的代码。我们将使用 CLR 的托管功能,让.NET 自己与操作系统进行交互。

## 4. 垃圾回收

托管代码最重要的一个功能是垃圾回收(garbage collection)。这种.NET 方法可确保应用程序不再使用某些内存时,就会完全释放这些内存。在.NET 推出以前,这项工作主要由程序员负责,代码中

的几个简单错误会把大块内存分配到错误的地方，使这些内存神秘失踪。这通常意味着计算机的速度逐渐减慢，最终导致系统崩溃。

.NET 垃圾回收会定期检查计算机内存，从中删除不再需要的内容。它不设置时间帧，可能一秒钟内会进行上千次的检查，也可能几秒钟才检查一次，或者随时进行检查，但一定会进行检查。

这里要给程序员一些提示。因为在不可预知的时间执行这项工作，所以在设计应用程序时，必须留意这一点。需要许多内存才能运行的代码应自己执行这样的检查，而不是坐等垃圾回收，但这不像听起来那样难。

5. 把它们组合在一起

在继续学习之前，先总结一下上述创建.NET 应用程序所经历的步骤：

- (1) 使用某种.NET 兼容语言(如 C#)编写应用程序代码，如图 1-1 所示。
- (2) 把代码编译为 CIL，存储在程序集中，如图 1-2 所示。



图 1-1

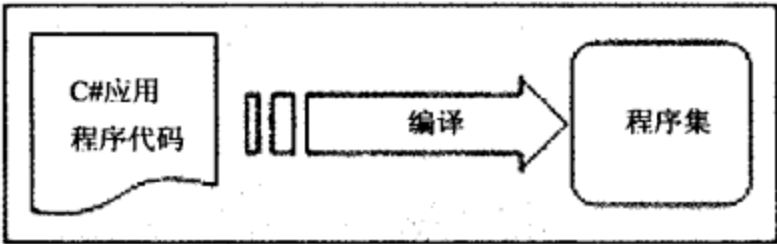


图 1-2

- (3) 在执行代码时(如果这是一个可执行文件，就自动运行，或者在其他代码使用它时运行)，首先必须使用 JIT 编译器将代码编译为本机代码，如图 1-3 所示。

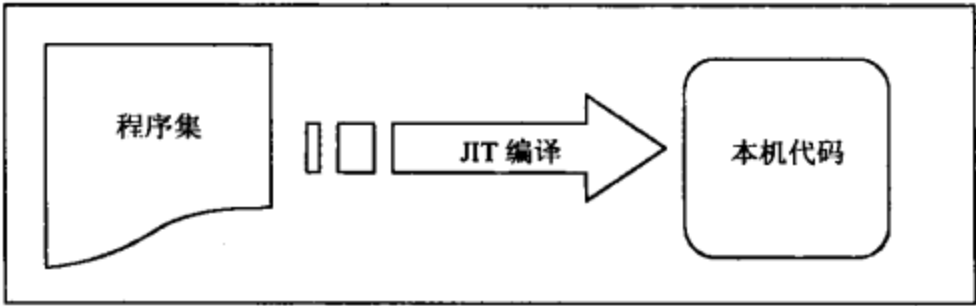


图 1-3

- (4) 在托管的 CLR 环境下运行本机代码，以及其他应用程序或进程，如图 1-4 所示。

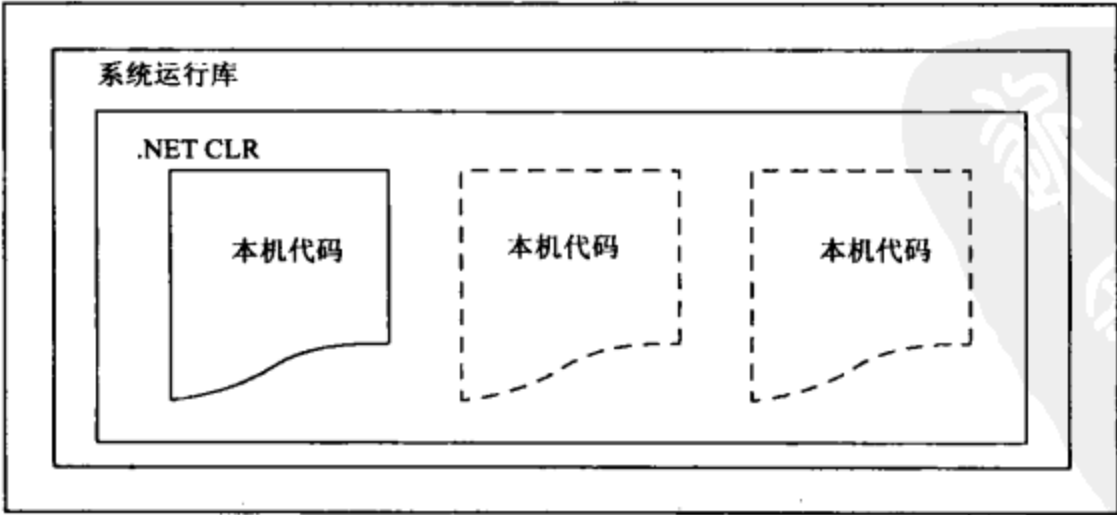


图 1-4



## 6. 链接

在上述过程中还有一点要注意。在第(2)步中编译为 CIL 的 C#代码不一定包含在单独文件中，可以把应用程序代码放在多个源代码文件中，再把它们编译到一个程序集中。这个过程称为链接(linking)，是非常有用的。原因是处理几个较小的文件比处理一个大文件要简单得多。可以把逻辑上相关的代码分解到一个文件中，以便单独进行处理，这也更易于在需要时找到特定的代码块，让开发小组把编程工作分解为一些可管理的块，让每个人编写一小块代码，而不会破坏已编写好的代码部分或其他人正在处理的部分。

## 1.2 C#的含义

如上所述，C#是可用于创建要运行在.NET CLR 上的应用程序的语言之一，它从 C 和 C++语言演化而来，是 Microsoft 专门为使用.NET 平台而创建的。因为 C#是近期发展起来的，所以吸取了以往的教训，考虑了其他语言的许多优点，并解决了它们的问题。

使用 C#开发应用程序比使用 C++简单，因为其语法比较简单。但是，C#是一种强大的语言，在 C++中能完成的任务几乎都能利用 C#完成。如前所述，C#中与 C++高级功能等价的功能(例如直接访问和处理系统内存)，只能在标记为“不安全(unsafe)”的代码中使用。这个高级编程技术存在潜在威胁(正如它的名称所暗示的)，因为它可能覆盖系统中重要的内存块，导致严重后果。因此，本书不讨论这个问题。

C#代码常比 C++略长一些。这是因为 C#是一种类型安全的语言(与 C++不同)。在外行人看来，这表示一旦为某个数据指定了类型，就不能转换为另一个不相关的类型。所以，在类型之间转换时，必须遵守严格的规则。执行相同的任务时，用 C#编写的代码通常比用 C++编写的代码长。但 C#代码更健壮，调试起来也比较简单，.NET 始终可以随时跟踪数据的类型。在 C#中，不能完成诸如“把 4 字节的内存放在这个数据中，使之有 10 个字节长，并把它解释为 X”等的任务，但这并不是一件坏事。

C#只是用于.NET 开发的一种语言，但它是最好的一种语言。C#的优点是，它是唯一彻头彻尾为.NET Framework 设计的语言，是在移植到其他操作系统上的.NET 版本中使用的主要语言。要使诸如 VB.NET 的语言尽可能类似于其以前的语言，且仍遵循 CLR，就不能完全支持.NET 代码库的某些功能，至少需要不常见的语法。但 C#能使用.NET Framework 代码库提供的每种功能。.NET 的最新版本还对 C#语言进行了几处改进，满足了开发人员的要求，使之更强大。

### 1.2.1 用 C#能编写什么样的应用程序

如前所述，.NET Framework 没有限制应用程序的类型。C#使用的是.NET Framework，所以也没有限制应用程序的类型。这里仅讨论几种常见的应用程序类型。

- **Windows 应用程序** 这些应用程序(如 Microsoft Office)具有我们很熟悉的 Windows 外观和操作方式，使用.NET Framework 的 Windows Forms 模块就可以简便地生成这种应用程序。Windows Forms 模块是一个控件库，其中的控件(例如，按钮、工具栏、菜单等)可以用于建立 Windows 用户界面(UI)。另外，还可以使用 Windows Presentation Foundation (WPF)建立 Windows 应用程序，WPF 提供了更大的灵活性和更卓越的功能。

- **Web 应用程序** 它们是一些 Web 页面，可以通过任何 Web 浏览器查看。.NET Framework 包括一个动态生成 Web 内容的强大系统，允许进行个性化和实现安全性等。这个系统叫作 Active Server Pages .NET(ASP.NET)，我们可以使用 C#通过 Web Forms 创建 ASP.NET 应用程序。还可以使用 Silverlight 编写在浏览器内部运行的应用程序。
- **Web 服务** 这是创建各种分布式应用程序的激动人心的新方式，使用 Web 服务可以通过 Internet 虚拟交换数据。无论使用什么语言创建 Web 服务，也无论 Web 服务驻留在什么系统上，都使用一样简单的语法。对于更高级的功能，还可以创建 Windows Communication Foundation(WCF)服务。

这些类型也需要某种形式的数据库访问，这可以通过 .NET Framework 的 Active Data Objects .NET(ADO.NET)部分、ADO.NET Entity Framework 或 C#的 LINQ(Language Integrated Query)功能来实现。也可以使用许多其他资源，例如，创建联网组件、输出图形、执行复杂数学任务的工具。

### 1.2.2 本书中的 C#

本书第 I 部分介绍 C# 语言的语法和用法，但不过分强调 .NET Framework。这是必需的，因为我们不能没有一点儿 C# 编程基础就使用 .NET Framework。首先介绍一些比较简单的内容，把较复杂的面向对象编程(Object-Oriented Programming, OOP)论题放在基础知识的后面论述。假定读者没有一点儿编程的知识，这些是首要的规则。

学习了基础知识后，本书还将介绍如何开发更复杂、更有用的应用程序。本书的第 II 部分将讨论 Windows Forms 编程，第 III 部分将研究 Web 应用程序和 Web 服务编程，第 IV 部分将讲述数据访问(对数据库、文件系统和 XML 数据的访问)，第 V 部分将介绍其他一些有趣的 .NET 论题。

## 1.3 Visual Studio 2010

本书使用 Visual Studio 2010(VS)或 Visual C# 2010 Express(VCE)开发工具进行所有的 C#编程，包括简单的命令行应用程序，乃至比较复杂的项目类型。VS 不是开发 C#应用程序所必需的开发工具或集成开发环境(IDE)，但使用它可以使任务更简单一些。可以在基本的文本编辑器(如常见的记事本)中处理 C#源代码文件，再使用命令行应用程序(是 .NET Framework 的一部分)把代码编译到程序集中。但是，为什么不使用功能完备的 IDE 呢？

下面列出的是一些使 VS 成为 .NET 开发首选工具的功能。

- VS 可以自动执行编译源代码的步骤，同时可以完全控制重写它们时应使用的任何选项。
- VS 文本编辑器为 VS 支持的语言(包括 C#)量身定制，这样就可以智能检测错误，在输入代码时给出合适的推荐代码，这个功能称为 IntelliSense。
- VS 包括 Windows Forms、Web Forms 及其他应用程序的设计器，允许 UI 元素的简单拖放设计。
- 在 C#中，许多类型的项目都可以用已有的“样板”代码来创建，不需要从头开始。各种代码文件通常已经准备好了，减少了从头开始一个项目所用的时间。对于“Starter Kit”项目类型来说尤其如此，该项目类型可以以功能全面的应用程序为基础进行开发。一些 Starter Kit 项目类型包含在 VS 安装程序中，还可以联机使用更多的该项目类型。

- VS 包括几个可自动执行常见任务的向导，其中的很多任务可以在已有的文件中添加合适的代码，在某些情况下，您甚至不需要考虑语法的正确性。
- VS 包含许多强大的工具，可以显示项目中的元素并允许在其中导航，这些元素可以是 C# 源代码文件，也可以是其他资源，例如位图图像或声音文件。
- 除了在 VS 中编写应用程序比较简单外，还可以创建部署项目，以便为客户提供代码，并使客户方便地完成安装。
- 在开发项目时，VS 允许使用高级调试技巧，例如，能在代码中一次调试一条指令，并监视应用程序的状态。

C#还有许多功能，希望读者能掌握它们！

### 1.3.1 Visual Studio 2010 Express 产品

除了 Visual Studio 2010 之外，Microsoft 还提供了几个更简单的开发工具，称为 Visual Studio 2010 Express 产品。可以在 <http://www.microsoft.com/express> 上免费获得它们。

其中两个产品是 Visual C# 2010 Express 和 Visual Web Developer 2010 Express，它们都可以创建所需的几乎所有 C# 应用程序。在功能上它们都是 VS 的删节版本，但外观和操作方式是一样的。尽管它们提供了 VS 的许多功能，但缺少一些重要的功能；不过我们仍可以在学习本书的过程中使用它们。

本书尽可能使用 VCE 开发 C# 应用程序，仅在需要某些功能时才使用 VS。当然，如果有 VS，就无需使用 Express 产品。

### 1.3.2 解决方案

在使用 VS 或 VCE 开发应用程序时，可以通过创建解决方案来完成。在 VS 和 VCE 术语中，解决方案不仅仅是一个应用程序，它还包含项目，可以是 Windows Forms 项目和 Web Form 项目等。但是，解决方案可以包含多个项目，这样，即使相关的代码最终在硬盘上的多个位置编译为多个程序集，也可以把它们组合到一个地方。

这是非常有用的，因为它可以处理“共享”代码(这些代码放在 GAC 中)，同时，应用程序也使用这段共享代码。在使用唯一的开发环境时，调试代码是非常容易的，因为可以在多个代码块中单步调试指令。

## 1.4 小结

本章简要介绍了 .NET Framework，并讨论了如何轻松创建各种强大的应用程序。还探讨了把用 C# 等语言编写的代码转换为可运行的应用程序所需要做的工作，以及使用在 .NET 公共语言运行库下运行的托管代码有什么优点。

本章还阐述了 C# 的实质，以及它与 .NET Framework 的关系，描述了进行 C# 开发时所使用的工具——Visual Studio 2010 和 Visual C# 2010 Express。

第 2 章将介绍如何运行一些 C# 代码，介绍基础知识，并集中讨论 C# 语言本身，而不是过多地讨论 IDE 的工作原理。



1.5 本章要点

主 题	重 要 概 念
.NET Framework 基础	.NET Framework 是 Microsoft 最新的开发平台，目前的版本是 4。它包括一个公共类型系统(CTS)和一个公共语言运行库(CLR)。.NET Framework 应用程序使用面向对象的编程(OOP)的方法编写，通常包含托管代码。托管代码的内存管理由.NET 运行库处理，其中包括垃圾回收
.NET Framework 应用程序	用.NET Framework 编写的应用程序首先编译为 CIL。在执行应用程序时，JIT 把 CIL 编译为本机代码。应用程序编译后，把不同的部分链接到包含 CIL 的程序集中
C#基础	C#是包含在.NET Framework 中的一种语言，它是以前的语言(如 C++)的一种演变，可以用于编写任意应用程序，包括网站和 Windows 应用程序
集成开发环境 (IDE)	可以在 Visual Studio 2010 中用 C#编写任意类型的.NET 应用程序，还可以在免费的、但功能稍弱的 Express 产品系列(包括 Visual C# Developer Express)中用 C#创建.NET 应用程序。这两种 IDE 都使用解决方案，解决方案可以包含多个项目



# 第 2 章

## 编写 C# 程序

### 本章内容:

---

- Visual Studio 2010 和 Visual C# 2010 Express Edition 的基础知识
- 如何编写简单的控制台应用程序
- 如何编写 Windows Form 应用程序

第 1 章用一定的篇幅讨论了 C# 是什么, 它是如何适应 .NET Framework 的, 现在就该编写一些代码了。本书主要使用 Visual Studio 2010 (VS) 和 Visual C# 2010 Express (VCE), 所以首先介绍这些开发环境的一些基础知识。

VS 是一个庞大复杂的产品, 可能会使初学者望而生畏, 但使用它创建简单的应用程序是非常容易的。在本章开始使用 VS 时, 不需要了解许多知识, 就可以编写 C# 代码。本书的后面将介绍 VS 能执行的更复杂的操作, 现在仅介绍基础知识。

从 VCE 入手要简单得多。在本书的前面部分, 所有示例都是在这个 IDE 中编写。但是如果选择使用 VS, 所有工作都或多或少地以相同的方式完成。因此, 本章介绍这两个 IDE, 先介绍 VS。

介绍完 IDE 后, 将把两个简单的应用程序组合在一起。现在不要过多地考虑代码, 只要应用程序可以运行即可。在这些早期的示例中熟悉了应用程序的创建过程, 不久之后就会适应这个过程了。

下面要创建的第一个应用程序是一个简单的控制台应用程序。控制台应用程序没有使用图形化的 Windows 环境, 所以不需要考虑按钮、菜单、用鼠标指针进行的交互等, 而是在命令行窗口中运行应用程序, 用更简单的方式与其交互。

第二个应用程序是一个 Windows Forms 应用程序, 其外观和操作方式对 Windows 用户来说会非常熟悉, 而且该应用程序创建起来并不费力。但所需代码的语法比较复杂, 尽管在许多情况下, 并不需要考虑细节。

本书接下来的两个部分也使用这两种应用程序类型, 但开始时略微强调一下控制台应用程序。在学习 C# 语言时, 不需要了解 Windows 应用程序的其他灵活性能。控制台应用程序的简单性可以让我们集中精力学习语法, 而无需考虑应用程序的外观和操作方式。

## 2.1 开发环境

本节讨论 VS 和 VCE 开发环境，先介绍 VS。这些环境是类似的，无论使用哪个 IDE，都应了解这两个环境。

### 2.1.1 Visual Studio 2010

在第一次加载 VS 时，会立即显示一系列窗口以及一组菜单和工具栏图标，其中大多数窗口是空的。本书将使用大多数窗口，读者很快就会熟悉它们。

如果是第一次运行 VS，则屏幕上会显示一个首选项列表，如果用户使用过这个开发环境的旧版本，则可以在这里做出选择，这些选择会影响到很多方面，例如，窗口的布局、控制台窗口运行的方式等。所以应选择 **Visual C# Development Settings**，否则步骤就不像下面描述的这样了。注意，可用选项会随着安装 VS 时选择的选项而变化，但只要选择安装 C#，这个选项就是可用的。

如果不是第一次运行 VS，但选择了另一个选项，不必惊慌。为了把设置重置为 **Visual C# Development Settings**，只需导入它们即可。为此，单击 Tools 菜单上的 **Import and Export Settings** 选项，再选中 **Reset All Settings** 选项，如图 2-1 所示。

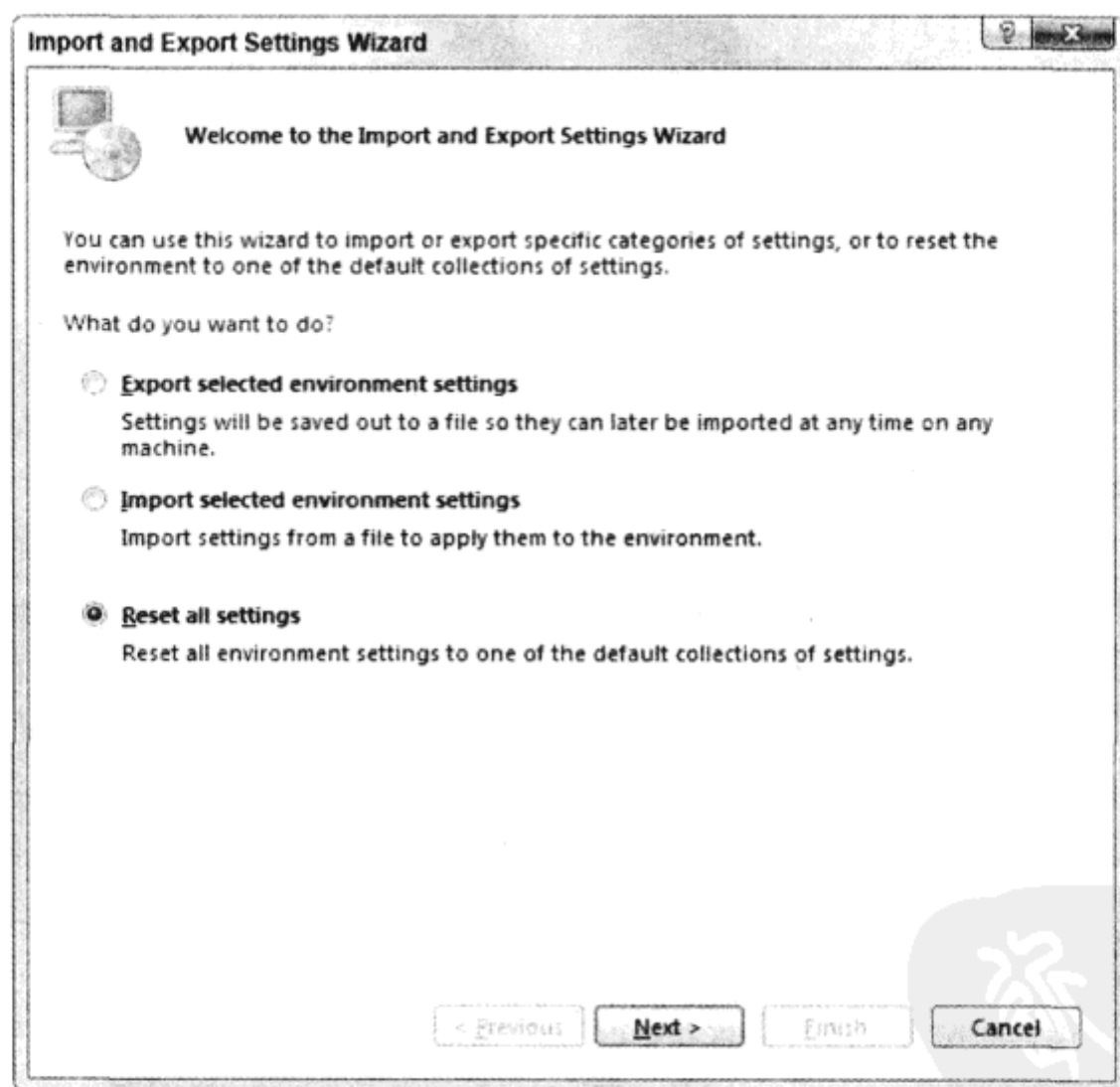


图 2-1

单击 Next 按钮，选择是否要在继续之前保存已有的设置。如果对设置进行了定制，就保存设置，否则就选择 No 按钮，再次单击 Next 按钮。在下一个对话框中，选择 **Visual C# Development Settings** 选项，如图 2-2 所示。可用的选项可能会变化。



图 2-2

最后单击 Finish 按钮，应用设置。

VS 环境布局是完全可定制的，但默认设置很适合我们。在 C# Developer Settings 设置下，其布局如图 2-3 所示。

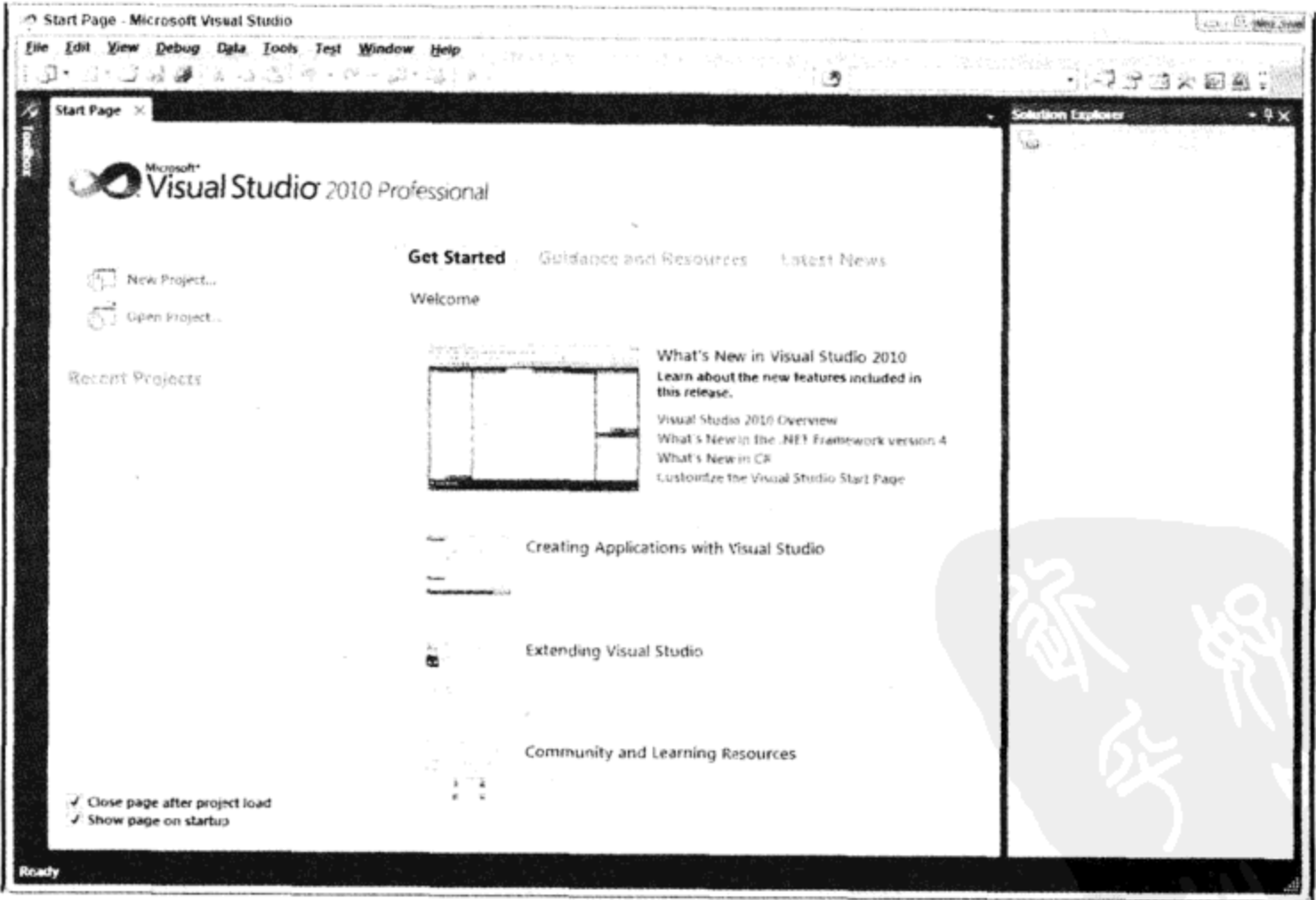


图 2-3



在 VS 启动时, 选项卡会默认显示一个介绍性的 **Start Page**, 并显示所有的代码。这个窗口可以包含许多文档, 每个文档都有一个选项卡, 单击文件名, 就可以在文件之间切换。这个窗口也具有其他功能: 它可以显示设计用于项目、纯文本文件和 HTML 的图形用户界面以及各种内置于 VS 的工具。本书将陆续介绍它们。

在主窗口的上面, 有工具栏和 VS 菜单。这里有几个不同的工具栏, 其功能包括保存和加载文件, 生成和运行项目, 以及调试控件等。在需要使用这些工具栏时将会讨论它们。

下面简要描述 VS 的最常用功能:

- 把鼠标指针放在 **Toolbox** 上, 就会显示 **Toolbox** 工具栏, 它们提供了 Windows 应用程序的用户界面构件等条目。另一个选项卡 **Server Explorer** 也在这里显示(通过 **View Server Explorer** 菜单项也可以选择它), 它包含其他许多功能, 例如, 访问数据源、服务器设置和服务等。
- **Solution Explorer** 窗口显示当前加载的解决方案的信息。如上一章所述, 解决方案是包含一个或多个项目及其配置的 Visual Studio 术语。**Solution Explorer** 窗口显示了解决方案中项目的各种视图, 例如, 项目中包含了什么文件, 这些文件中又包含了什么内容。
- **Solution Explorer** 窗口之下可以显示 **Properties** 窗口, 该窗口没有显示在图 2-3 中。稍后会看到这个窗口, 因为它只在处理项目时才出现(也可以使用 **View | Properties Window** 菜单项切换它)。这个窗口提供了更详细的项目内容视图, 允许另外配置单独元素。例如, 使用这个窗口可以改变 Windows 窗体中按钮的外观。
- 另一个非常重要的窗口也未出现在图 2-3 中: **Error List** 窗口。这个窗口可以使用 **View | Error List** 菜单项打开, 它显示了错误、警告和其他与项目有关的信息。这个窗口会持续不断地更新, 但其中一些信息只有在编译项目时才出现。

这似乎要理解很多东西, 但不必担心, 过不了多久就习惯了。下面首先建立第一个示例项目, 它将使用上面介绍的许多 VS 元素。



VS 还可以显示许多其他窗口, 它们都包含许多信息, 有许多功能。其中的一些窗口与上面提及的窗口在相同的位置上, 使用选项卡可以切换它们。本书的后面会介绍其中的许多窗口, 在详细介绍 VS 环境时, 还会发现更多的窗口。

### 2.1.2 Visual C# 2010 Express Edition

使用 VCE, 您不必考虑设置的改变。显然, 这个产品并不适用于 VB 编程, 所以无需考虑相应的设置。第一次启动 VCE 时, 会显示一个非常类似于 VS 的屏幕, 如图 2-4 所示。

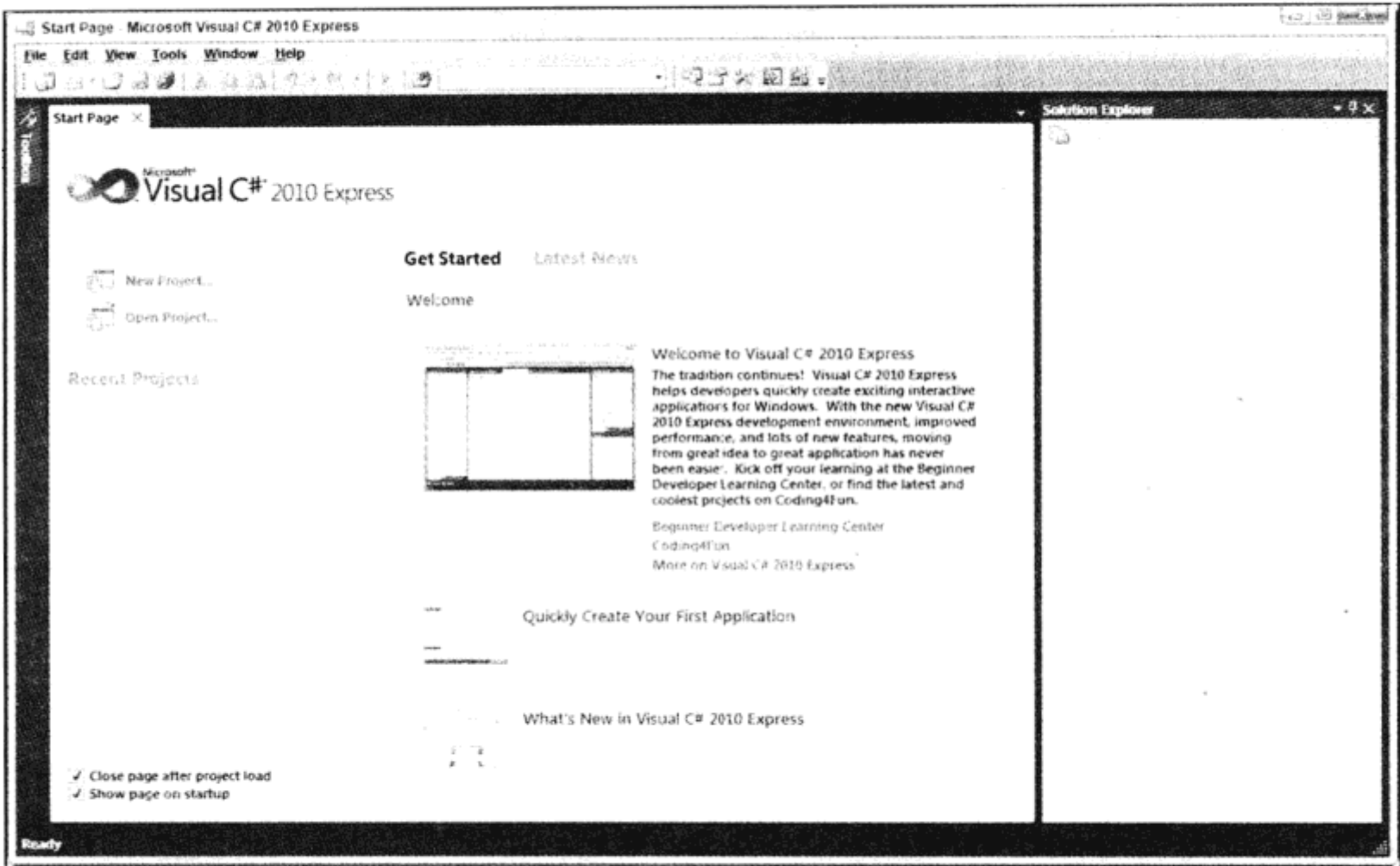


图 2-4

## 2.2 控制台应用程序

本书将频繁使用控制台应用程序，特别是开始时要使用这类应用程序，所以下面创建一个简单的控制台应用程序。这个示例包含了用于 VS 和 VCE 的指令。

**试一试：创建一个简单的控制台应用程序**

(1) 在 VS 中选择 **File | New | Project...**菜单项，或者在 VCE 中选择 **File | New Project...**，创建一个新的控制台应用程序项目，如图 2-5 和 2-6 所示。

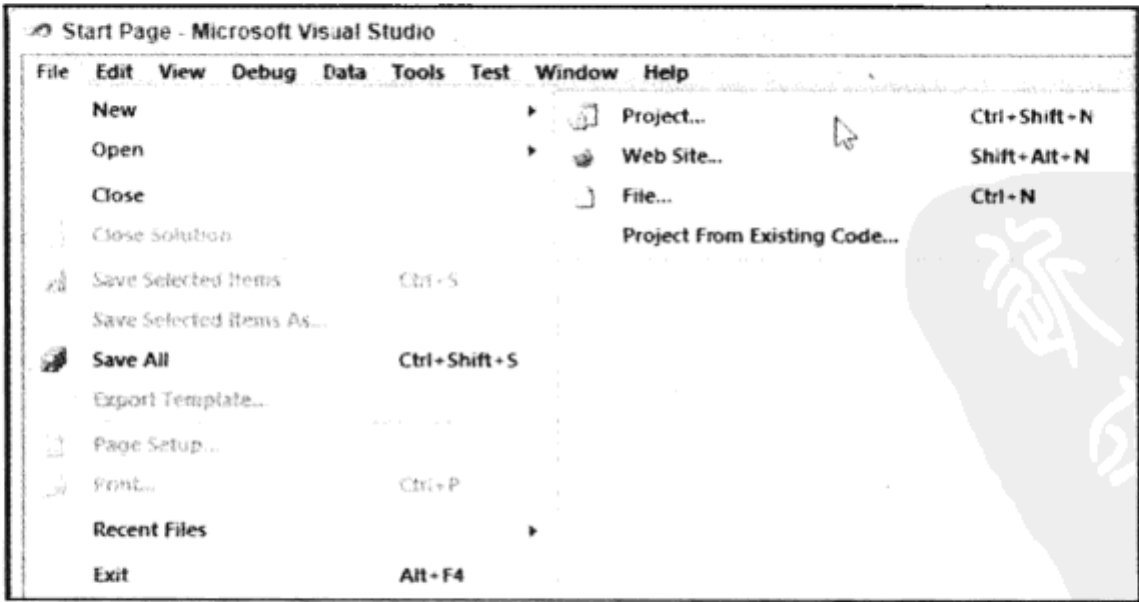


图 2-5

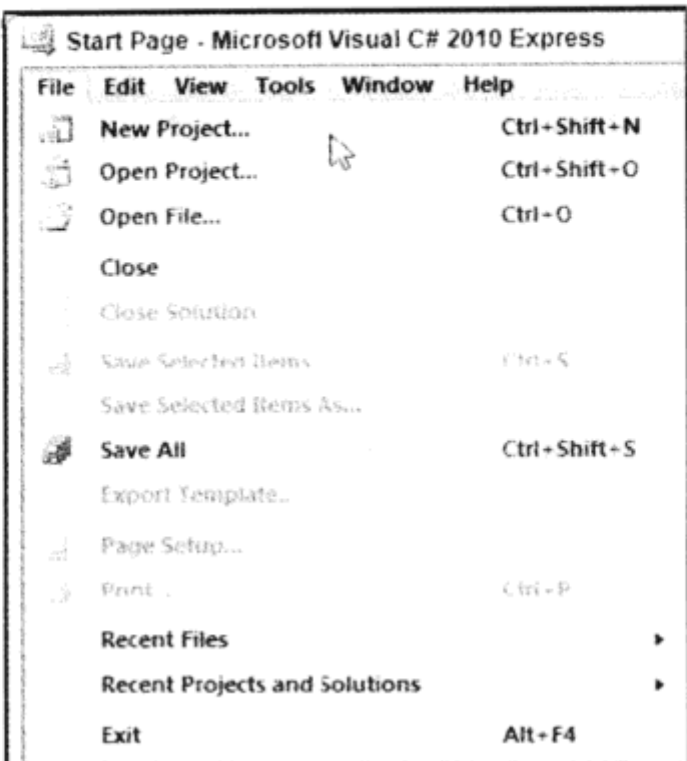


图 2-6

(2) 在 VS 显示窗口的 Installed Templates 窗格中选择 Visual C# 节点,在中间窗格中选择 Console Application 项目类型,如图 2-7 所示。在 VCE 中,只需在 Templates 窗格中选择 Console Application 项目类型,如图 2-8 所示。在 VS 中把 Location 文本框改为 C:\BegVCSharp\Chapter02(如果该目录不存在,会自动创建)。在 VS 和 VCE 中,Name 文本框中的默认文本(ConsoleApplication1)和其他设置不变,如图 2-7 和 2-8 所示。



图 2-7

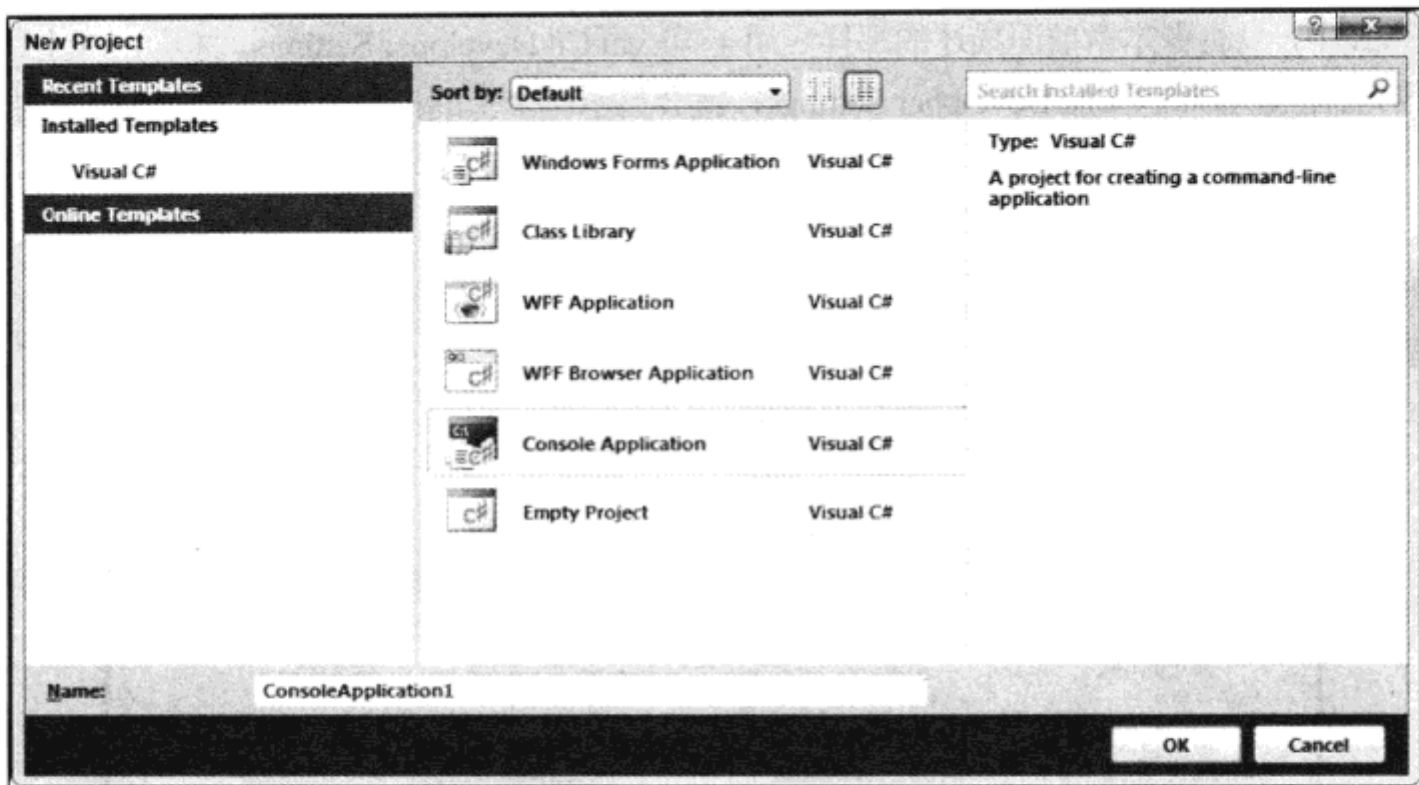


图 2-8

- (3) 单击 OK 按钮。
- (4) 如果使用的是 VCE，在初始化项目后，单击工具栏上的 Save All 按钮或选择 File 菜单中的 Save All 选项，将 Location 字段设置为 C:\BegVCSharp\Chapter02，单击 Save 按钮。
- (5) 初始化项目后，在主窗口显示的文件里添加如下代码行：



```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Output text to the screen.
            Console.WriteLine("The first app in Beginning C# Programming!");
            Console.ReadKey();
        }
    }
}
```

代码段 ConsoleApplication1\Program.cs

- (6) 选择 Debug | Start Debugging 菜单项。稍后就应看到如图 2-9 所示的结果。

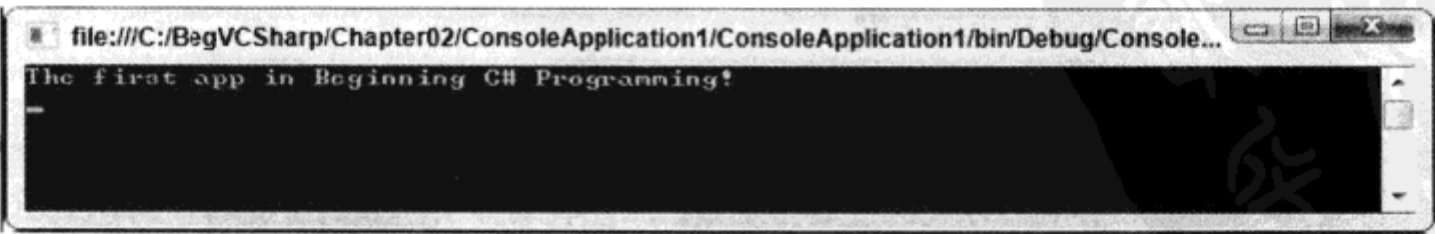


图 2-9

- (7) 按下任意键，退出应用程序(首先需要单击控制台窗口，以激活它)。



在 VS 中, 只有像本章前面描述的那样应用了 Visual C# Developer Settings, 才会出现上述显示。例如, 若应用了 Visual Basic Developer Settings, 就会显示一个空的控制台窗口, 应用程序的输出结果显示在 Immediate 窗口中。在这种情况下, Console.ReadKey() 代码也会失败, 显示一个错误。如果遇到这个问题, 本书中所有示例的最佳解决方案是应用 Visual C# Developer Settings, 这样读者看到的结果才会与书中显示的相同。如果问题没有解决, 可以打开 Tools | Options 对话框, 取消对 Debugging | Redirect all Output Window text to the Immediate Window 选项的选择, 如图 2-10 所示。

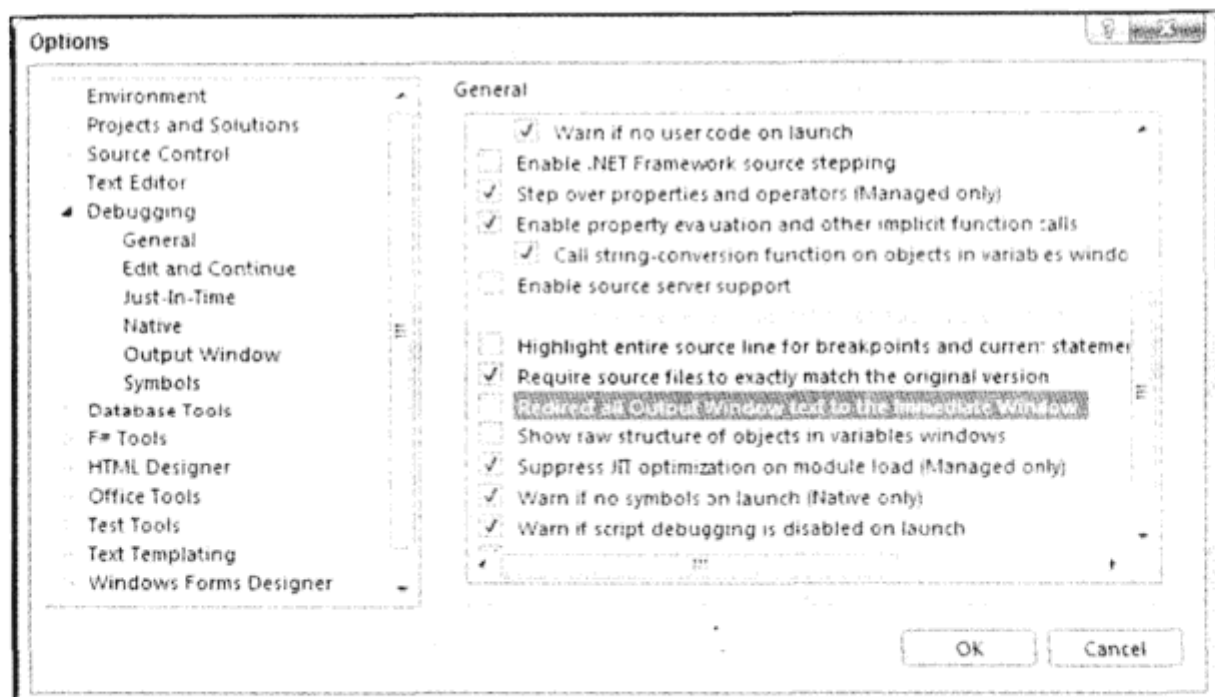


图 2-10

### 示例的说明

现在不仔细研究这个项目中使用的代码, 而关心如何使用开发工具来启动和运行代码。显然, VS 和 VCE 做了许多工作, 简化了编译和执行代码的过程。执行这些简单的步骤还有多种方式。例如, 创建一个新项目可以像前面那样使用 File | New Project... 菜单项, 也可以按下 Ctrl+Shift+N 组合键, 还可以单击工具栏上相应的图标。

同样, 也可以用多种方式编译和执行代码。上面使用的方法是选择 Debug | Start Debugging 菜单项, 也可以按下快捷键(F5), 或者使用工具栏中的图标。使用 Debug | Start Without Debugging 菜单项(也可以按下 Ctrl+F5 组合键)还可以以非调试模式运行代码, 使用 Build | Build Solution 或 F6 可以编译项目但不运行它(打开或关闭调试功能)。注意, 执行项目但不调试, 或者生成项目可以使用工具栏中的图标进行, 只是这些图标在默认情况下没有显示出来。编译好代码后, 在 Windows Explorer 中运行生成的 .exe 文件, 就可以执行代码。也可以在命令提示窗口中执行, 为此, 应打开一个命令提示窗口, 把目录改为 C:\BegVCSharp\Chapter02\ConsoleApplication1\ConsoleApplication1\bin\Debug\, 键入 ConsoleApplication1, 并按下回车键。



在以后的示例中, 我们仅说明“创建一个新的控制台项目”或“执行代码”, 用户可以选择自己喜欢的方式执行这些步骤。除非特别声明, 否则所有的代码都应在启用调试的情况下运行。另外, 本书中的“启动”、“执行”和“运行”等术语都可以互换使用, 示例后面的讨论总是假定已经退出了示例中的应用程序。

控制台应用程序会在执行完毕后立即终止,如果直接通过 IDE 运行它们,就无法看到运行的结果。为了解决上例中的这个问题,使用

```
Console.ReadKey();
```

告诉代码在结束前等待按键。后面的示例将多次使用这种技术。前面创建了一个项目,现在详细讨论开发环境中的各个组成部分。

### 2.2.1 Solution Explorer

首先要讨论的窗口是屏幕右上角的 Solution Explorer,它在 VS 和 VCE 中是相同的(除非特别说明,否则本章分析的所有窗口在 VS 和 VCE 中都是相同的)。这个窗口默认设置为自动隐藏,在该窗口可见时,单击其图钉图标可以把它停靠在屏幕的一条边上。Solution Explorer 窗口与另一个有用的窗口 Class View 在相同的位置上,使用 View | Class View 菜单项就可以显示 Class View 窗口。图 2-11 显示了展开所有节点的这两个窗口(在窗口停靠时,单击窗口底部的选项卡,就可以切换它们)。

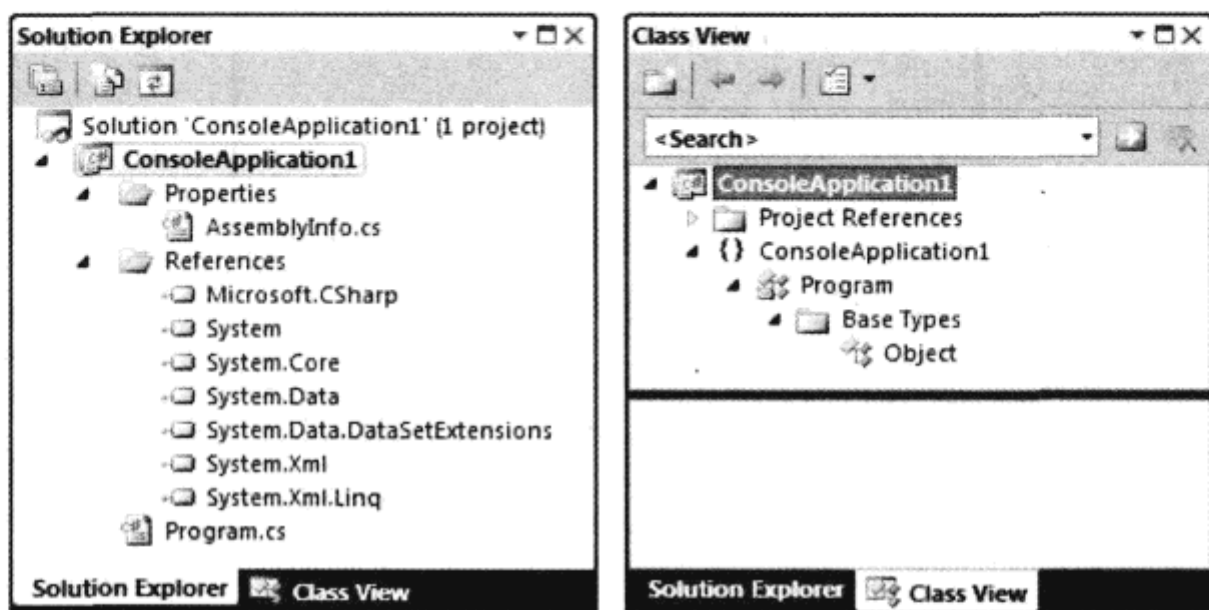


图 2-11



在 VCE 中,只有打开 Expert Settings,才能使用 Class View 窗口。通过 Tools | Settings | Expert Settings 菜单项可以打开 Expert Settings。

Solution Explorer 视图显示了组成 ConsoleApplication1 项目的文件,包括我们在其中添加代码的文件 Program.cs、另一个代码文件 AssemblyInfo.cs 和几项引用。



所有 C#文件都使用.cs 文件扩展名。

此时不需要考虑 AssemblyInfo.cs 文件,它包含项目中目前我们不需要关心的其他信息。

使用这个窗口可以改变主窗口中显示的代码,方法是双击.cs 文件,或右击这些文件并选择 View Code,或选中它们,单击窗口顶部的工具栏按钮。还可以对这些文件执行其他操作,例如,重新命

名它们，或从项目中删除它们等。在该窗口中还可以显示其他类型的文件，例如，项目资源(资源是项目使用的文件，这些文件可能不是 C#文件，如位图图像和声音文件等)。可以通过同一界面处理它们。

References 项包含项目中使用的一个.NET 库列表，这个列表在后面介绍，因为标准引用很适于初学者使用。视图 Class View 显示了项目的另一种视图，可以用于查看刚才创建的代码结构。本书后面将介绍代码结构，现在选择显示 Solution Explorer 视图。单击这些窗口中的文件或其他图标，Properties 窗口的内容就会发生相应变化，如图 2-12 所示。

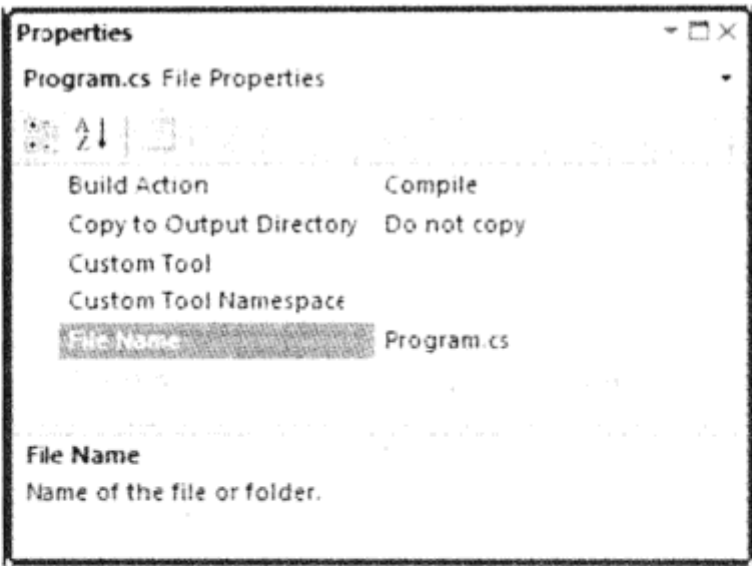


图 2-12

2.2.2 Properties 窗口

使用 View | Properties Window 菜单项就可以打开 Properties 窗口。这个窗口显示了在上述窗口中所选的项的其他信息。例如，选择项目中的 Program.cs 文件，就会显示如图 2-12 所示的视图。这个窗口还显示了其他选中项的信息，例如，用户界面组件(参见本章的“Windows Forms 应用程序”一节)。

通常在 Properties 窗口中对项目的改变会直接影响代码，添加代码行，或改变文件中的内容。对于一些项目来说，通过这个窗口来操作与手动修改代码所花的时间是相同的。

2.2.3 Error List 窗口

当前 Error List 窗口(View | Error List)没有显示什么有趣的信息，这是因为应用程序没有错误。但是这的确是一个非常有用的窗口。下面进行测试，从上一节添加的一行代码中删除分号。过一会儿，就会看到如图 2-13 所示的结果。

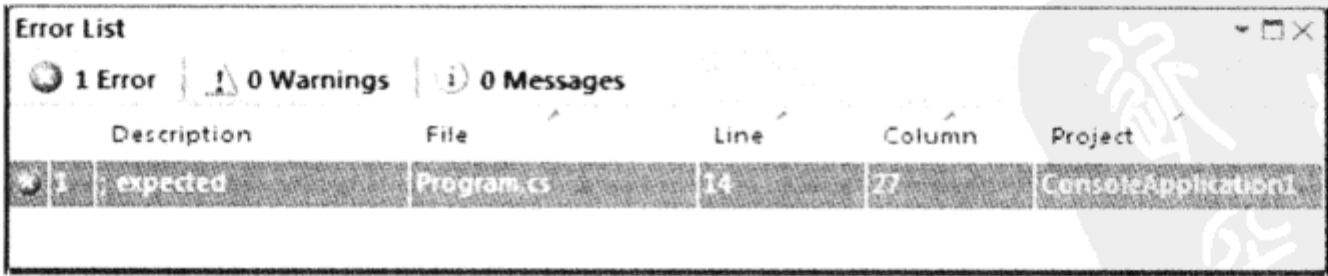


图 2-13

这次项目不会编译。



第3章介绍 C#语法后,你就会明白分号在大多数代码行的末尾,它是必不可少的。

这个窗口有助于根除代码中的错误,因为它会跟踪我们的工作,编译项目。如果双击该窗口中显示的错误,光标就会跳到源代码中出现错误的地方(如果包含错误的源文件没有打开,将被打开),这样就可以快速更正错误。代码中有错误的一行会出现红色的波浪线,便于我们快速扫描源代码,找出错误。

注意错误的位置用一个行号来指定。在默认情况下,行号不会显示在 VS 文本编辑器中,但其实有必要显示它。为此,需要单击 Tools | Options 菜单项,选中 Options 对话框中的 Line numbers 复选框。该复选框位于 Text Editor | C# | General 类别中,如图 2-14 所示。

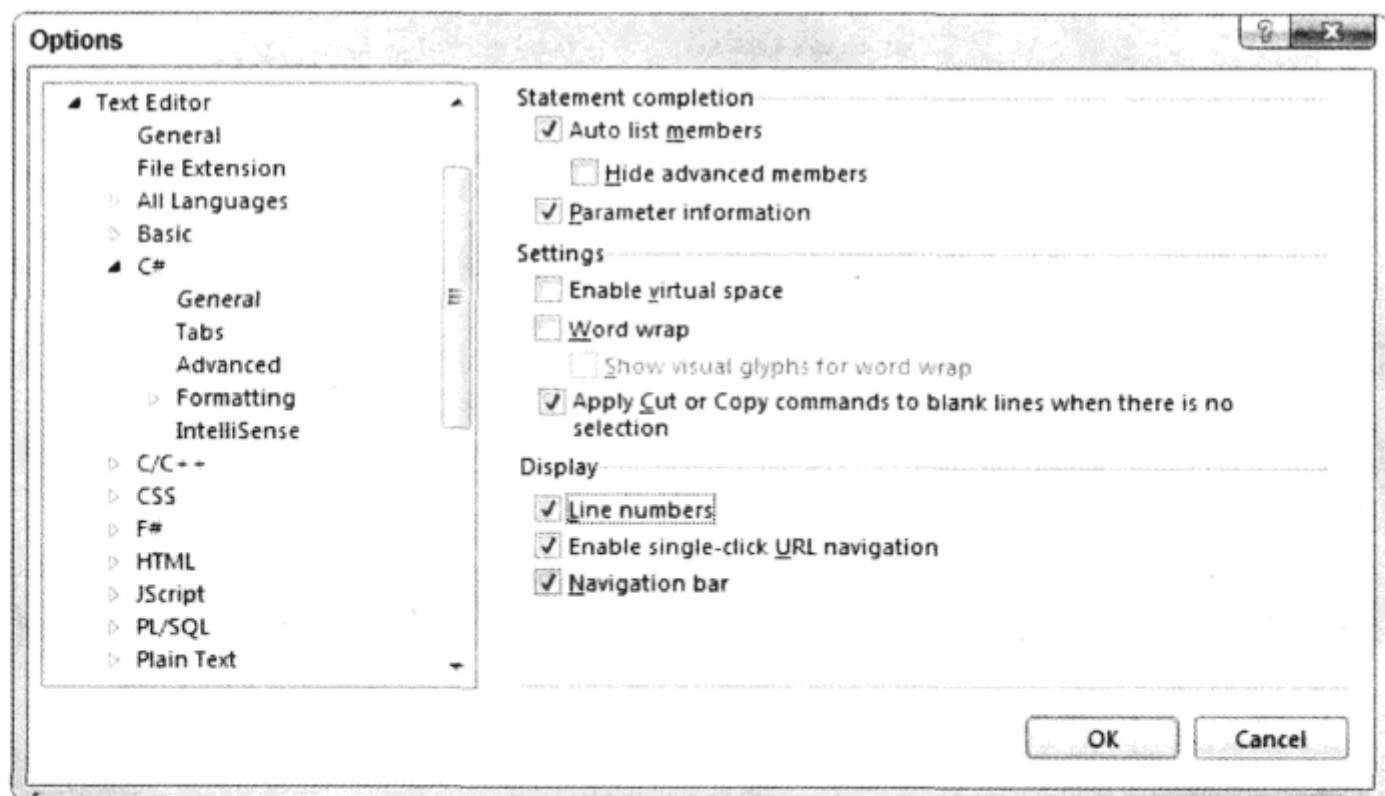


图 2-14



在 VCE 中,只有选择 Show All Settings 才能使这个选项可用,选项列表与图 2-14 略有不同。

这个对话框中包含许多有用的选项,本书将使用其中的几个。

## 2.3 Windows Forms 应用程序

通常,如果将代码当作 Windows 应用程序的一部分运行来描述代码,要比通过控制台窗口或命令提示符简单一些。下面用用户界面构件来组合一个用户界面。

下面的示例介绍建立用户界面的基础知识,说明如何启动和运行 Windows 应用程序,但并不详



细讨论应用程序实际完成的工作。本书的后面会详细研究 Windows 应用程序。

试一试：创建一个简单的 Windows 应用程序

(1) 在以前的位置(C:\BegVCSharp\Chapter02，如果使用的是 VCE，就在创建该项目后把它保存在这个位置)创建一个类型为 Windows Forms Application(VS 或 VCE)的新项目，其默认名称是 WindowsFormsApplication1。如果使用的是 VS，而第一个项目仍处于打开状态，则应选择 Create New Solution 选项来启动一个新解决方案，这些设置如图 2-15 和图 2-16 所示。

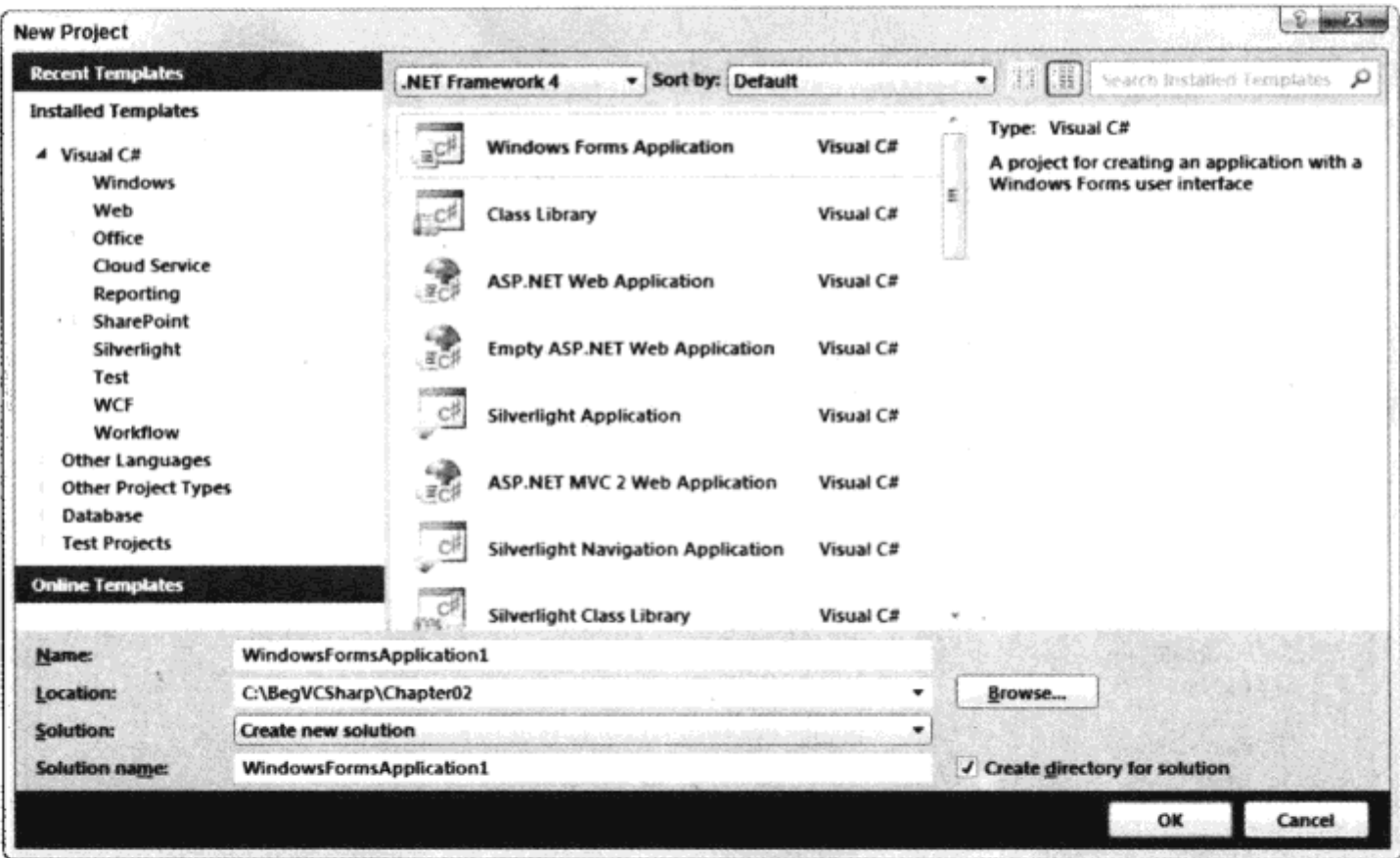


图 2-15

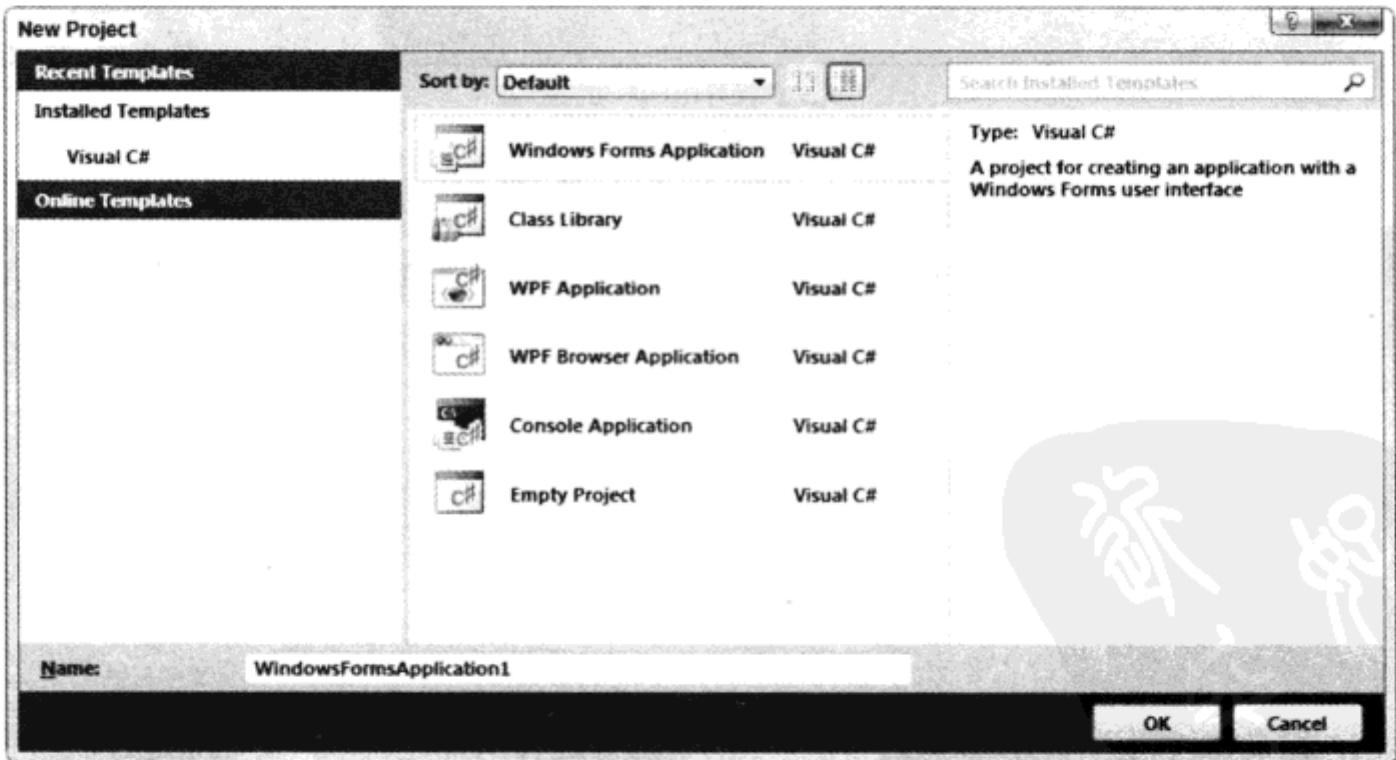


图 2-16

(2) 单击 OK 按钮，创建项目后，应该会看到一个空白的 Windows 窗体。把鼠标指针移到屏幕

左边的 Toolbox 栏上, 然后移到 All Windows Forms 选项卡上的 Button 选项, 在该选项上双击, 就会在应用程序的主窗体(Form1)上添加一个按钮。

(3) 双击刚才添加到窗体中的按钮。

(4) 现在应显示 Form1.cs 中的 C#代码。进行如下修改(为了简短起见, 这里只显示了文件中的部分代码):

可从  
wrox.com  
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("The first Windows app in the book!");
}
```

代码段 WindowsFormsApplication1\Form1.cs

(5) 运行应用程序。

(6) 单击显示出来的按钮, 打开一个消息对话框, 如图 2-17 所示。



图 2-17

(7) 单击 OK。像每个标准 Windows 应用程序那样, 单击右上角的 X 图标, 退出应用程序。

#### 示例的说明

IDE 又一次自动完成了许多工作, 使我们能不费吹灰之力就可以完成一个实用的 Windows 应用程序的创建。刚才创建的应用程序与其他窗口的行为方式相同——可以移动、重新设置其大小、最小化等。我们不必编写任何代码, 它就可以工作。我们添加的按钮也是这样。双击按钮, IDE 就知道我们想添加一些代码, 当运行应用程序时, 用户单击该按钮, 就执行我们已经编写好的代码。只要提供了这段代码, 就可以得到按钮单击的所有功能。

当然, Windows 应用程序不仅限于带有按钮的普通窗体。如果看看从中选择 Button 选项的工具栏, 就会看到一整套用户界面构件, 其中一些用户可能很熟悉。本书在其他地方将使用其中的大多数用户界面构件, 它们使用起来都非常简单, 可以节省许多时间和精力。

应用程序的代码在 Form1.cs 中, 看起来并不比上一节提供的代码复杂多少, Solution Explorer 窗口中其他文件的代码也不太复杂。开发环境生成的代码在默认情况下是隐藏的, 它们都与窗体上控件的布局有关。这就是为什么可以在主窗口的设计视图中查看代码的原因, 而设计视图正是这些布局代码的可视化转换。按钮就是可以使用的一种控件, 同样, 也可以使用 Toolbox 上 Windows Forms 部分中的其他 UI 构件。

下面把按钮作为一个控件示例，详细解释一下。使用主窗口中的选项卡，切换回窗体的设计视图，单击按钮，选择它。此时，屏幕右下角的 **Properties** 窗口就会显示按钮控件的属性(控件的属性非常类似于上例中文件的属性)，确保应用程序当前并未运行，向下滚动到 **Text** 属性，它目前设置为 **button1**，把其值改为 **Click Me**，如图 2-18 所示。

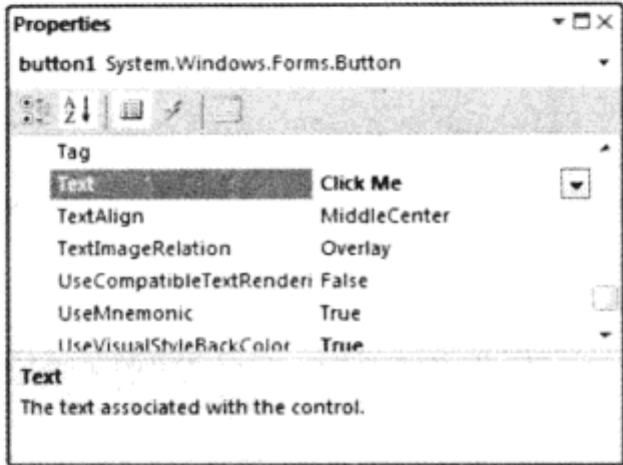


图 2-18

在 **Form1** 中，按钮上的文本应反映这个变化。

这个按钮有许多属性，从按钮颜色和大小简单格式，到某些模糊设置(如数据绑定设置，它可以建立与数据库的联系)，应有尽有。如上例所述，改变属性通常会直接改变代码，这也不例外。但如果切换回 **Form1.cs** 的代码视图，就会发现代码没有变化。

要查看修改过的代码，需要查看上面提及的隐藏代码。为了查看包含代码的文件，需要在 **Solution Explorer** 窗口中扩展 **Form1.cs** 节点。打开 **Form1.Designer.cs** 节点，双击这个文件，就可以查看其中的内容。

匆匆一瞥可能注意不到代码中哪些地方反映了按钮属性的改变，这是因为 C# 代码中处理窗体上控件的布局和格式化的部分被隐藏了(毕竟，如果有了结果的图形显示，就几乎不需要查看代码了)。

**VS** 和 **VCE** 使用代码突出显示系统来显示这部分 C# 代码，如图 2-19 所示。

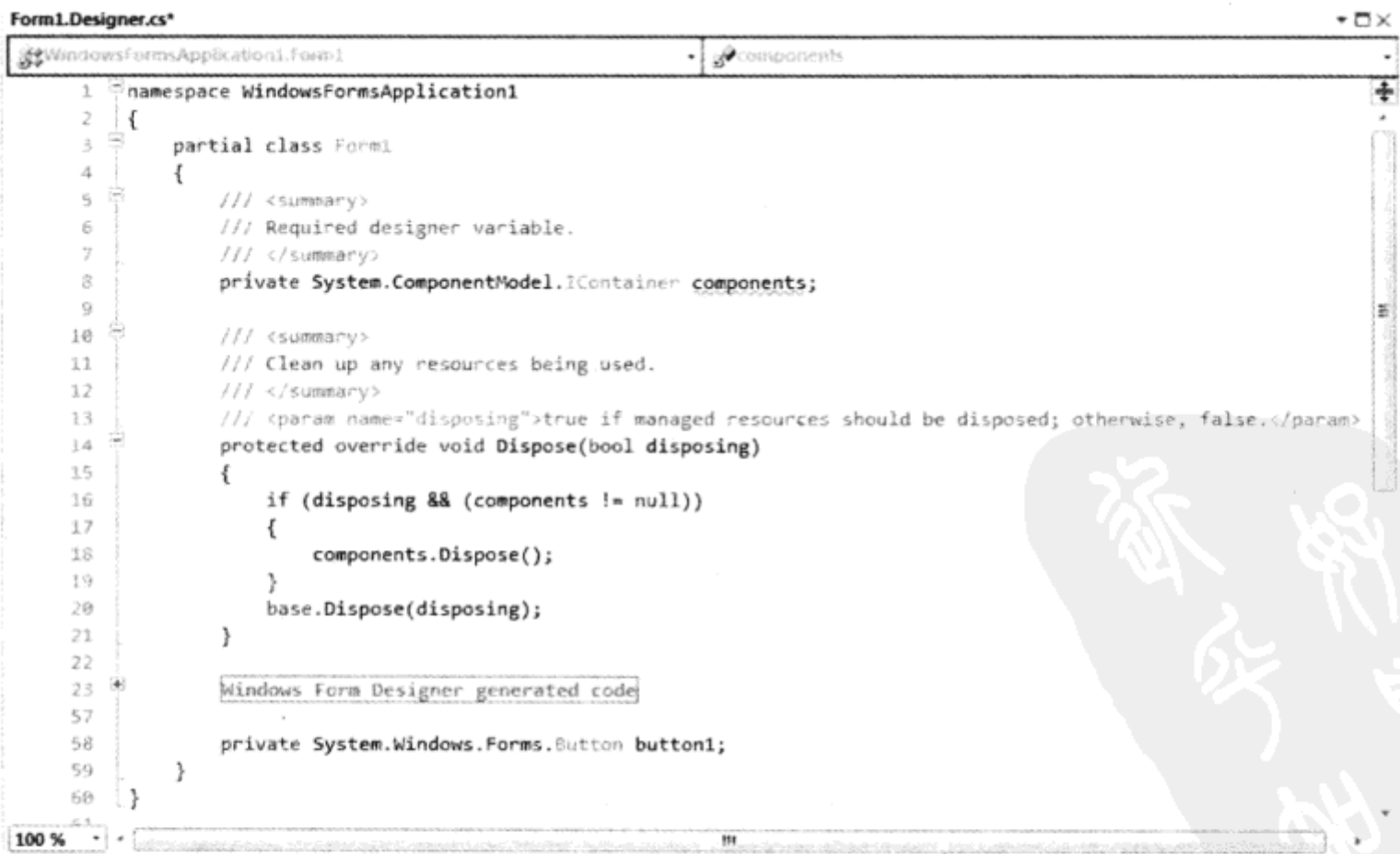


图 2-19

注意在代码的左边(如果已显示行号,就在行号的旁边),有一些灰色的线条,其上有+和-号的方框。这些方框可用于展开和折叠代码区域。靠近文件的底部,有一个带+号的方框,它对应代码主体中的“Windows Form Designer generated code”。这个标签的基本含义是“这里有一些 VS 生成的代码,用户不需知道”。但如果愿意,也可以进行查看,并通过修改按钮属性来了解所做的事情。只要单击带有+号的方框,代码就会显示出来,在某处应该会看到如下代码:

```
this.button1.Text = "Click Me";
```

不必过多地考虑这里使用的语法,我们可以看到在 Properties 窗口中键入的文本直接显示在代码中。

在编写代码时,这种突出显示代码的方式带来了极大的方便。我们可以展开和折叠其他许多区域(而不仅仅是正常情况下隐藏的代码段)。查看一本书的目录,有助于快速了解该书的内容;查看一组折叠的代码段,则非常便于浏览大量的 C#代码。

2.4 小结

本章介绍了本书后面所使用的一些工具,快速浏览了 Visual Studio 2010 和 Visual C# 2010 Express 开发环境,并使用它们建立了两种类型的应用程序。其中比较简单的是控制台应用程序,它足以满足我们的大多数需要,便于我们集中精力学习 C#编程的基础知识。Windows 应用程序比较复杂,但其可视化程度比较高,对于习惯了 Windows 环境的人来说,使用起来也比较直观。

知道如何创建简单的应用程序,就可以真正开始学习 C#了。本书后面的章节将介绍 C#的基本语法和程序结构,之后讨论更高级的面向对象方法。学习了这些内容后,就可以开始了解如何使用 C#访问.NET Framework 的功能了。

在后面的章节中,除非特别说明,否则指令都是针对 VCE 的,但如本章所述,这些指令很容易修改,以用于 VS,读者可以使用自己喜欢的任何 IDE。

2.5 本章要点

主 题	重 要 概 念
Visual Studio 2010 设置	本书需要在第一次运行 VS 时选择 C# Development Settings 选项,或者重置它们
控制台应用程序	控制台应用程序是简单的命令行应用程序,本书主要用它演示技术。在 VS 或 VCE 中创建新项目时,使用 Console Application 模板就会创建新的控制台应用程序。要在调试模式下运行项目,可以使用 Debug   Start Debugging 菜单项或者按下 F5
IDE 窗口	项目内容显示在 Solution Explorer 窗口中。选中项的属性显示在 Properties 窗口中。错误显示在 Error List 窗口中
Windows 窗体应用程序	Windows 窗体应用程序具备标准桌面应用程序的外观和操作方式,包括最大化、最小化和关闭应用程序等大家熟悉的图标。它们是在 New Project 对话框中用 Windows Forms 模板创建的





# 第 3 章

## 变量和表达式

本章内容:

- C#的基本语法
- 变量及其用法
- 表达式及其用法

要想高效地使用 C#, 就一定要理解创建计算机程序时真正需要做些什么。计算机程序最基本的描述也许是一系列处理数据的操作, 即使对于最复杂的示例, 例如 Microsoft Office 套装软件之类的大型多功能 Windows 应用程序, 这个论述也正确。应用程序的用户虽然看不到它们, 但这些操作总是在后台进行。

为了进一步解释它, 考虑一下计算机的显示单元。我们常常比较熟悉屏幕上的内容, 很难不把它想像为“移动的图片”。但实际上, 我们看到的仅是一些数据的显示结果, 其最初的形式是存储在计算机内存中的 0 和 1 数据流。因此我们在屏幕上进行的任何操作, 无论是移动鼠标指针, 单击图标, 或在字处理器上输入文本, 都会改变内存中的数据。

当然, 还可以利用一些较简单的情形来说明这一点。如果使用计算器应用程序, 就要提供数字, 对这些数字执行操作, 就像用纸和笔计算数字一样, 但使用程序会快得多。

如果计算机程序是对数据执行操作, 则说明我们需要以某种方式来存储数据, 需要某些方法来处理它们。这两种功能是由变量和表达式提供的, 本章将探究它们的含义。

在开始之前, 应该首先了解一下 C#编程的基本语法, 因为我们需要一个环境来学习使用 C#语言中的变量和表达式。

### 3.1 C#的基本语法

C#代码的外观和操作方式与 C++和 Java 非常类似。初看起来, 其语法可能比较混乱, 不像书面英语和其他语言那么直观。但实际上, 在 C#编程中, 使用的样式是比较清晰的, 不用花太多力气就可以编写出便于阅读的代码。

与其他语言的编译器不同，C#编译器不考虑代码中的空格、回车符或 tab 字符(这些字符统称为空白字符)。这样格式化代码时就有很大的自由度，但遵循某些规则将有助于阅读代码。

C#代码由一系列语句组成，每个语句都用一个分号来结束。因为空白被忽略，所以一行可以有多个语句，但从可读性的角度来看，通常在分号的后面加上回车符，这样就不会在一行上放置多个语句了。但语句放在多个行上是可以的(也比较常见)。

C#是一种块结构的语言，所有的语句都是代码块的一部分。这些块用花括号来界定(“{”和“}”)，代码块可以包含任意多行语句，或者根本不包含语句。注意花括号字符不需要附带分号。

所以，简单的 C#代码块如下所示：

```
{
    <code line 1, statement 1>;
    <code line 2, statement 2>
    <code line 3, statement 2>;
}
```

其中<code line x, statement y>部分并非真正的 C#代码，而是用这个文本作为 C#语句的占位符。在这段代码中，第 2、3 行代码是同一个语句的一部分，因为在第 2 行的末尾没有分号。

下面的简单示例还使用了缩进格式，提高了 C#代码的可读性。这是一个标准做法，实际上在默认情况下 VS 会自动缩进代码。一般情况下，每个代码块都有自己的缩进级别，即它向右缩进了多少。代码块可以互相嵌套(即块中可以包含其他块)，而被嵌套的块要缩进得多一些。

```
{
    <code line 1>;
    {
        <code line 2>;
        <code line 3>;
    }
    <code line 4>;
}
```

前面代码行的续行通常也要缩进得多一些，如上面第一个示例中的第 3 行代码所示。



在能通过 Tools | Options 访问的 VCE 或 VS Options 对话框中，显示了 VCE 用于格式化代码的规则。在 Text Editor | C# | Formatting 节点的子类别下，包含了其中的很多规则。此处的大多数设置都反映了还没有讲述的 C#部分，但如果以后要修改设置，以更适合自己的个性化样式，就可以回过头来看看这些设置。在本书中，为了简洁起见，所有代码段都使用默认设置来格式化。

当然，这种样式并不是强制的。但如果不使用它，读者在阅读本书时会很快陷入迷茫之中。

在 C#代码中，另一个常见的语句是注释。注释并非严格意义上的 C#代码，但代码最好有注释。注释就是自解释，即给代码添加描述性文本(用英语、法语、德语、外蒙古语等)，编译器会忽略这些内容。在开始处理冗长的代码段时，注释可用于为正在进行的工作添加提示，例如“这行代码要求用户输入一个数字”，或“这段代码由 Bob 编写”。

C#添加注释的方式有两种。可以在注释的开头和结尾放置标记，也可以使用一个标记，其含义是“这行代码的其余部分是注释”。在 C#编译器忽略回车符的规则中，后者是一个例外，但这是一

种特殊情况。

要使用第一种方式标记注释，可以在注释开头加上`/*`字符，在末尾加上`*/`字符。这些注释符号可以在单独一行上，也可以在不同的行上，注释符号之间的所有内容都是注释。注释中唯一不能输入的是`*/`，因为它会被看作注释结束标记。所以下面的语句是正确的：

```
/* This is a comment */

/* And so...

    ... is this! */
```

但以下语句会产生错误：

```
/* Comments often end with "*/" characters */
```

注释结束符号后的内容(`*/`后面的字符)会被当作 C# 代码，因此产生错误。

另一种添加注释的方法是用`//`开始一个注释，在其后可以编写任何内容，只要这些内容在一行上即可。下面的语句是正确的：

```
// This is a different sort of comment.
```

但下面的语句会失败，因为第二行代码会解释为 C# 代码：

```
// So is this,
    but this bit isn't.
```

这类注释可用于语句的说明，因为它们都放在一行上：

```
<A statement>;          // Explanation of statement
```

前面讲过，有两种给 C# 代码添加注释的方法。但在 C# 中，还有第三类注释，严格地说，这是语法的扩展。它们都是单行注释，用三个`/`符号来开头，而不是两个。

```
/// A special comment
```

正常情况下，编译器会忽略它们，就像其他注释一样，但可以配置 VS，在编译项目时，提取这些注释后面的文本，创建一个特殊格式的文本文件，该文件可用于创建文档说明书。为了创建文档说明书，注释必须遵循 XML 文档的规则。本书不讨论这个主题，如果读者有时间，这是一个很值得学习的内容。

特别要注意的一点是，C# 代码是区分大小写的。与其他语言不同，必须使用正确的大小写形式输入代码，因为简单地用大写字母代替小写字母会中断项目的编译。看看下面这行代码，它在第 2 章的第一个示例中使用：

```
Console.WriteLine("The first app in Beginning C# Programming!");
```

C# 编译器能理解这行代码，因为 `Console.WriteLine()` 命令的大小写形式是正确的。但是，下面的语句都不能工作：

```
console.WriteLine("The first app in Beginning C# Programming!");
CONSOLE.WRITELINE("The first app in Beginning C# Programming!");
Console.Writeline("The first app in Beginning C# Programming!");
```

这里使用的大小写形式是错误的，所以 C#编译器不知道我们要做什么。幸好，VCE 在代码的键入方面提供了许多帮助，在大多数情况下，它都知道(程序也知道)我们要做什么。在键入代码的过程中，VS 会推荐用户可能要使用的命令，并尽可能纠正大小写问题。

## 3.2 C#控制台应用程序的基本结构

下面看看第 2 章的控制台应用程序示例(ConsoleApplication1)，并研究一下它的结构。其代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Output text to the screen.
            Console.WriteLine("The first app in Beginning C# Programming!");
            Console.ReadKey();
        }
    }
}
```

可以立即看出，上一节讨论的所有语法元素这里都有。其中有分号、花括号、注释和适当的缩进。目前看来，这段代码中最重要的部分如下所示：

```
static void Main(string[] args)
{
    // Output text to the screen.
    Console.WriteLine("The first app in Beginning C# Programming!");
    Console.ReadKey();
}
```

在运行控制台应用程序时，就会运行这段代码，更确切地讲，是运行花括号中的代码块。如前所述，注释行不做任何事情，包含它们只为了简明而已。其他两行代码在控制台窗口中输出一些文本，并等待一个响应。但目前我们还不需要关心它的具体机制。

这里要注意一下如何实现第 2 章介绍的代码突出显示功能，这对于 Windows 应用程序来说比较重要，因为它是一个非常有用的特性。要实现该功能，需要使用 `#region` 和 `#endregion` 关键字，来定义可以展开和折叠的代码区域的开头和结尾。例如，可以修改为 ConsoleApplication1 生成的代码，如下所示：

```
#region Using directives

using System;
```



```
using System.Collections.Generic;
using System.Linq;
using System.Text;
#endregion
```

这样就可以把这些代码行折叠为一行，以后要查看其细节时，可以再次展开它。这里包含的 `using` 语句和其下的 `namespace` 语句将在本章后面予以解释。



以#开头的任意关键字实际上都是一个预处理指令，严格地说并不是 C# 关键字。除了这里描述的 `#region` 和 `#endregion` 关键字之外，其他关键字都相当复杂，用法也比较专业。所以，这是一个读者通读全书后才能探究的主题。

现在不必考虑示例中的其他代码，因为本书前几章仅解释 C# 的基本语法，至于应用程序进行 `Console.WriteLine()` 调用的具体方式，则不在我们的考虑之列。以后会阐述此附加代码的重要性。

### 3.3 变量

如前所述，变量关系到数据的存储。实际上，可以把计算机内存中的变量看作架子上的盒子。在这些盒子中，可以放入一些东西，再把它们取出来，或者只是看看盒子里是否有东西。变量也是这样，数据可放在变量中，可以从变量中取出数据或查看它们。

尽管计算机中的所有数据事实上都是相同的东西(一组 0 和 1)，但变量有不同的内涵，称为类型。下面再使用盒子来类比，盒子有不同的形状和尺寸，某些东西只能放在特定的盒子中。建立这个类型系统的原因是，不同类型的数据需要用不同的方法来处理。变量限定为不同的类型，可以避免混淆。例如，组成数字图片的 0 和 1 序列与组成声音文件的 0 和 1 序列，其处理方式是不同的。

要使用变量，需要声明它们。即给变量指定名称和类型。声明变量后，就可以把它们用作存储单元，存储所声明的数据类型的数据。

声明变量的 C# 语法是指定类型和变量名，如下所示：

```
<type> <name>;
```

如果使用未声明的变量，代码将无法编译，但此时编译器会告诉我们出现了什么问题，所以这不是一个灾难性错误。另外，使用未赋值的变量也会产生一个错误，编译器会检测出这个错误。

可以使用的变量类型是无限多的。其原因是可以自己定义类型，存储各种复杂数据。尽管如此，总有一些数据类型是每个人都要使用的，例如，存储数值的变量。因此，我们应了解一些简单的预定义类型。

#### 3.3.1 简单类型

简单类型就是组成应用程序中基本构件的类型，例如，数值和布尔值(true 或 false)。简单类型与复杂类型不同，没有子类型或特性。大多数简单类型都是存储数值的，初看起来有点奇怪，肯定只需要一种类型来存储数值吗？

数值类型过多的原因是在计算机内存中，把数字作为一系列的 0 和 1 来存储的机制。对于整数值，用一定的位(单个数字，可以是 0 或 1)来存储，用二进制格式来表示。以 N 位来存储的变量可以表示任何介于 0 到  $(2^N-1)$ 之间的数。大于这个值的数因为太大，所以不能存储在这个变量中。

例如，有一个变量存储了 2 位，在整数和表示该整数的位之间的映射应如下所示：

0 = 00  
1 = 01  
2 = 10  
3 = 11

如果要存储更多数字，就需要更多的位(例如，3 位可以存储 0~7 的数)。

这样得到的结论是要存储每个可以想像得到的数，就需要非常多的位，这并不适合 PC。即使可以用足够多的位来表示每一个数，变量使用这些位来存储它，其效率也非常低下，例如，只需要存储从 0~10 之间的数(因为存储器被浪费了)。其实 4 位就足够了，可以用相同的内存空间存储这个范围内的更多数值。

相反，许多不同的整数类型可以用于存储不同范围的数值，占用不同的内存空间(至多 64 位)，这些类型如表 3-1 所示。

表 3-1

类 型	别 名	允 许 的 值
sbyte	System.SByte	在 -128~127 之间的整数
byte	System.Byte	在 0~255 之间的整数
short	System.Int16	在 -32 768~32 767 之间的整数
ushort	System.UInt16	在 0~65 535 之间的整数
int	System.Int32	在 -2 147 483 648~2 147 483 647 之间的整数
uint	System.UInt32	在 0~4 294 967 295 之间的整数
long	System.Int64	在 -9 223 372 036 854 775 808~9 223 372 036 854 775 807 之间的整数
ulong	System.UInt64	在 0~18 446 744 073 709 551 615 之间的整数



这些类型中的每一种都利用了 .NET Framework 中定义的标准类型。如第 1 章所述，使用标准类型可以在语言之间交互操作。在 C# 中这些类型的名称是 Framework 中定义的类型别名，表 3-1 列出了这些类型在 .NET Framework 库中的名称。

一些变量名称前面的“u”是 unsigned 的缩写，表示不能在这些类型的变量中存储负数，参见该表中的“允许的值”一列。

当然，还需要存储浮点数，它们不是整数。可以使用的浮点数变量类型有 3 种：float、double 和 decimal。前两种可以用  $\pm m \times 2^e$  的形式存储浮点数，m 和 e 的值因类型而异。Decimal 使用另一种形式： $\pm m \times 10^e$ 。这 3 种类型、其 m 和 e 的值，以及它们在实数中的上下限如表 3-2 所示。

表 3-2

类 型	别 名	m 的 最小值	m 的 最大值	e 的 最小值	e 的 最大值	近似的最小值	近似的最大值
float	System.Single	0	2 <sup>24</sup>	- 149	104	1.5 × 10 <sup>-45</sup>	3.4 × 10 <sup>38</sup>
double	System.Double	0	2 <sup>53</sup>	- 1075	970	5.0 × 10 <sup>-324</sup>	1.7 × 10 <sup>308</sup>
decimal	System.Decimal	0	2 <sup>96</sup>	- 26	0	1.0 × 10 <sup>-28</sup>	7.9 × 10 <sup>28</sup>

除了数值类型外，另外还有 3 种简单类型，如表 3-3 所示。

表 3-3

类 型	别 名	允 许 的 值
char	System.Char	一个 Unicode 字符，存储 0~65 535 之间的整数
bool	System.Boolean	布尔值：true 或 false
string	System.String	一组字符

注意组成 string 的字符数没有上限，因为它可以使用可变大小的内存。

布尔类型 bool 是 C#中最常用的一种变量类型，类似的类型在其他语言的代码中非常丰富。当编写应用程序的逻辑流程时，一个可以是 true 或 false 的变量有非常重要的分支作用。例如，考虑一下有多少问题可以用 true 或 false(或 yes 和 no)来回答。执行变量值之间的比较或检查输入的有效性就是后面使用布尔变量的两个编程示例。

介绍了这些类型后，下面用一个简短示例来声明和使用它们。在下面的示例中，要使用一些简单的代码来声明两个变量，给它们赋值，再输出这些值。

试一试：使用简单类型的变量

- (1) 在目录 C:\BegVCSharp\Chapter03 下创建一个新的控制台应用程序 Ch03Ex01。
- (2) 在 Program.cs 中添加如下代码：



```
static void Main(string[] args)
{
    int myInteger;
    string myString;
    myInteger = 17;
    myString = "\"myInteger\" is";
    Console.WriteLine("{0} {1}.", myString, myInteger);
    Console.ReadKey();
}
```

代码段 Ch03Ex01\Program.cs

- (3) 运行代码，结果如图 3-1 所示。

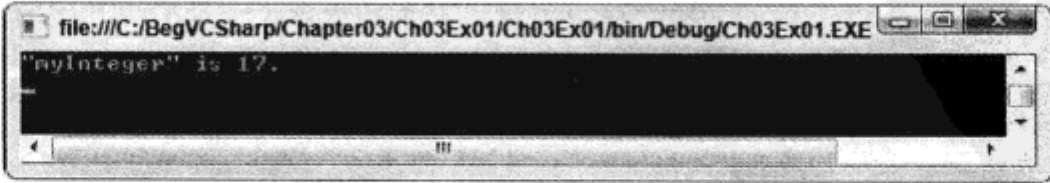


图 3-1

### 示例的说明

我们添加的代码完成了以下 3 项任务：

- 声明两个变量
- 给这两个变量赋值
- 将两个变量的值输出到控制台上

变量声明使用下述代码：

```
int myInteger;
string myString;
```

第一行声明一个类型为 `int` 的变量 `myInteger`，第二行声明一个类型为 `string` 的变量 `myString`。



变量的命名是有限制的，不能使用任意的字符序列。“变量的命名”将介绍变量的命名规则。

接下来的两行代码为变量赋值：

```
myInteger = 17;
myString = "\"myInteger\" is";
```

使用 `=` 赋值运算符(在本章的“表达式”一节中将详细介绍)给变量分配两个固定的值(在代码中称为字面值)。把整数值 17 赋给 `myInteger`，把字符串 `"myInteger"`(包括引号)赋给 `myString`。以这种方式给字符串赋予字面值时，必须用双引号把字符串括起来。因此，如果字符串本身包含双引号，就会出现错误，必须用一些表示这些字符的其他字符(即转义序列)来替代它们。本例使用序列 `\"` 来转义双引号：

```
myString = "\"myInteger\" is";
```

如果不使用这些转义序列，而输入如下代码：

```
myString = "myInteger is";
```

就会出现编译错误。

注意给字符串赋予字面值时，必须小心换行——C#编译器会拒绝分布在多行上的字符串字面值。如果要添加一个换行符，可以在字符串中使用换行符的转义序列，即 `\n`。例如，赋值语句：

```
myString = "This string has a \nline break.";
```

会在控制台视图中显示两行代码，如下所示：

```
This string has a
line break.
```

所有的转义序列都包含一个反斜杠符号，后跟一个字符组合(详见后面的内容)，因为反斜杠符号的这种用途，它本身也有一个转义序列，即两个连续的反斜杠 `\\`。

下面继续解释代码，还有一行没有说明：

```
Console.WriteLine("{0} {1}.", myString, myInteger);
```

它看起来类似于第一个示例中把文本写到控制台上的简单方法，但本例指定了变量。这里不算详细讨论这行代码。这是本书第 I 部分用于给控制台窗口输出文本的一种技巧，知道这一点就足够了。在括号中，有如下两项：

- 一个字符串
- 一个用逗号分隔的变量列表，这些变量的值将插入到输出字符串中

输出字符串是 "{0} {1}."，它们并没有包含有用的文本。可以看出，这并不是我们运行代码时希望看到的结果，其原因是：字符串实际上是插入变量内容的一个模板，字符串中的每对花括号都是一个占位符，包含列表中每个变量的内容。每个占位符(或格式字符串)用包含在花括号中的一个整数来表示。整数从 0 开始，每次递增 1，占位符的总数应等于列表中指定的变量数，该列表用逗号分隔开，跟在字符串后。把文本输出到控制台时，每个占位符就会用每个变量的值来替代。在上面的示例中，{0} 用第一个变量的值 `myString` 替换，{1} 用 `myInteger` 的内容来替换。

在后面的示例中，就使用这种给控制台输出文本的方式显示代码的输出结果。最后一行代码在前面的示例中也出现过，用于在程序结束前等待用户输入内容：

```
Console.ReadKey();
```

这里不详细探讨这行代码，但后面的示例会常常用到它。现在只需要知道，它暂停代码的执行，等待用户按下一个键。

### 3.3.2 变量的命名

如上一节所述，不能把任意序列的字符作为变量名。这并不像第一次听起来那样需要担心什么，因为这种命名系统仍是非常灵活的。

基本的变量命名规则如下：

- 变量名的第一个字符必须是字母、下划线( )或@。
- 其后的字符可以是字母、下划线或数字。

另外，有一些关键字对于 C# 编译器而言有特定的含义，例如前面出现的 `using` 和 `namespace` 关键字。如果错误地使用其中一个关键字，编译器会产生一个错误，我们马上就会知道出错了，所以不必担心。

例如，下面的变量名是正确的：

```
myBigVar
VAR1
_test
```

下列变量名有误：

```
99BottlesOfBeer
namespace
It's-All-Over
```

记住，C# 区分大小写，所以必须小心，不要忘了在声明变量时使用正确的大小写。在程序后面引用它们时，即使只有一个字母的大小写形式出错，都不能编译成功。其进一步的结果是得到多个变量，其名称仅有大小写的区别，例如，下面的变量都是不同的：



```
myVariable  
MyVariable  
MYVARIABLE
```

### 命名约定

变量名是比较常用的，所以有必要用一定的篇幅讨论几种要用到的变量名称。在开始前，要记住这是有争议的。多年以来，出现了不同的系统，一些开发人员拼命维护他们的个人系统。

最近，最流行的系统是所谓的 Hungarian 表示法。这个系统在所有的变量名上加上一个小写形式的前缀，表示其类型。例如，如果变量的类型是 `int`，就在其名称前加上 `i`(或 `n`)，如 `iAge`。使用这个系统，很容易看出各个变量是什么类型的。

更现代的语言如 C# 则很难实现这个系统。与前面介绍的所有类型一样，可以用一两个字母前缀表示变量的类型。但由于可以创建自己的类型，而且在 .NET Framework 中有上百种更复杂的类型，所以这种系统很快就失效了。在多人合作完成的项目中，不同的人很容易遇到易混淆的不同前缀，它们可能导致灾难性的后果。

开发人员现在认识到，最好根据变量的作用来命名它们。如果出现问题，就很容易确定变量的类型。在 VS 和 VCE 中，只需把鼠标指针在变量名上停留足够长的时间，就会弹出一个方框，指出该变量的类型。

目前，在 .NET Framework 名称空间中有两种命名约定，称为 `PascalCase` 和 `camelCase`。在名称中使用的大小写表示它们的用途。它们都应用到由多个单词组成的名称中，并指定名称中的每个单词除了第一个字母大写外，其余字母都是小写。在 `camelCase` 中，还有一个规则，即第一个单词以小写字母开头。

下面是 `camelCase` 变量名：

```
age  
firstName  
timeOfDeath
```

下面是 `PascalCase` 变量名：

```
Age  
LastName  
WinterOfDiscontent
```

Microsoft 建议：对于简单的变量，使用 `camelCase` 规则，而对于比较高级的命名则使用 `PascalCase`。最后，注意许多以前的命名系统常常使用下划线字符作为变量名中各个单词之间的分隔符，如 `yet_another_variable`。但这种用法现在已经淘汰了。

### 3.3.3 字面值

在前面的示例中，有两个字面值的示例：整数和字符串。其他变量类型也有相关的字面值，如表 3-4 所示。其中有许多涉及到后缀，即在字面值的后面添加一些字符，指定想要的类型。一些字面值有多种类型，在编译时由编译器根据它们的上下文确定其类型。

表 3-4

类 型	类 别	后 缀	示例/允许的值
bool	布尔	无	true 或 false
int, uint, long, ulong	整数	无	100
uint, ulong	整数	u 或 U	100U
long, ulong	整数	l 或 L	100L
ulong	整数	ul、uL、Ul、UL、lu、lU、Lu 或 LU	100UL
float	实数	f 或 F	1.5F
double	实数	无、d 或 D	1.5
decimal	实数	m 或 M	1.5M
char	字符	无	'a'或转义序列
string	字符串	无	"a...a", 可以包含转义序列

字符串的字面值

本章前面介绍了几个可以在字符串的字面值中使用的转义序列，表 3-5 是这些转义序列的完整列表，以便以后引用。

表 3-5

转 义 序 列	产生的字符	字符的 Unicode 值
\'	单引号	0x0027
\"	双引号	0x0022
\\	反斜杠	0x005C
\0	空	0x0000
\a	警告(产生蜂鸣)	0x0007
\b	退格	0x0008
\f	换页	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

表 3-5 中的“Unicode 值”列是字符在 Unicode 字符集中的 16 进制值。与上面一样，使用 Unicode 转义序列可以指定 Unicode 字符，该转义序列包括标准的\字符，后跟一个 u 和一个 4 位十六进制值 (例如，表 3-5 中 x 后面的 4 位数字)。

下面的字符串是等价的：

```
"Karli\'s string."  
"Karli\u0027s string."
```

显然，Unicode 转义序列还有更多用途。

也可以逐字地指定字符串，即两个双引号之间的所有字符都包含在字符串中，包括行末字符和需要转义的字符。唯一例外是双引号字符的转义，它们必须指定，以免结束字符串。为此，可以在该字符串之前加一个@字符：

```
@"Verbatim string literal."
```

可以采用一般方式指定这个字符串，但需要使用下面这种方法：

```
@"A short list:
item 1
item 2"
```

逐字指定的字符串在文件名中非常有用，因为文件名中大量使用了反斜杠字符。如果使用一般的字符串，就必须在字符串中使用两个反斜杠，例如：

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

而有了逐字指定的字符串字面值，这段代码就更便于阅读。下面的字符串与上面的等价：

```
@"C:\Temp\MyDir\MyFile.doc"
```



从本书的后面可以看出，字符串是引用类型，而本章中的其他类型都是值类型。所以，字符串也可以被赋予 null 值，即字符串变量不引用字符串。

### 3.3.4 变量的声明和赋值

快速回顾一下，前面使用变量的类型和名称来声明它们，例如：

```
int age;
```

然后用=赋值运算符给变量赋值：

```
age = 25;
```



变量在使用前，必须初始化。上面的赋值语句可以用作初始化语句。

这里还可以做两件事，用户可以在 C# 代码中看到。第一是同时声明多个类型相同的变量，方法是在类型的后面用逗号分隔变量名，如下所示：

```
int xSize, ySize;
```

其中 xSize 和 ySize 都声明为整数类型。

第二个技巧是在声明变量的同时为它们赋值，即把两行代码合并在一起：

```
int age = 25;
```

可以同时使用这两个技巧:

```
int xSize = 4, ySize = 5;
```

xSize 和 ySize 被赋予不同的值。注意下面的代码:

```
int xSize, ySize = 5;
```

其结果是 ySize 被初始化, 而 xSize 仅进行了声明, 在使用前仍需要初始化。

3.4 表达式

前面介绍了如何声明和初始化变量, 下面该处理它们了。C#包含许多进行这类处理的运算符。把变量和字面值(在使用运算符时, 它们都称为操作数)与运算符组合起来, 就可以创建表达式, 它是计算的基本构件。

运算符范围广泛, 有简单的, 也有非常复杂的, 其中一些可能只在数学应用程序中使用。简单的操作包括所有的基本数学操作, 例如+运算符是把两个操作数加在一起, 而复杂的操作包括通过变量内容的二进制表示来处理它们。还有专门用于处理布尔值的逻辑运算符, 以及赋值运算符, 如=运算符。

本章主要介绍数学和赋值运算符, 而逻辑运算符将在第 4 章中介绍, 主要论述控制程序流程的布尔逻辑。

运算符大致分为如下 3 类。

- 一元运算符, 处理一个操作数
- 二元运算符, 处理两个操作数
- 三元运算符, 处理三个操作数

大多数运算符都是二元运算符, 只有几个一元运算符和一个三元运算符, 即条件运算符(条件运算符是一个逻辑运算符, 详见第 4 章)。下面先介绍数学运算符, 它包括一元运算符和二元运算符。

3.4.1 数学运算符

有 5 个简单的数学运算符, 其中两个有二元和一元两种形式。表 3-6 列出了这些运算符, 并用一个简短示例来说明它们的用法, 以及使用简单的数值类型(整数和浮点数)时它们的结果。

表 3-6

运 算 符	类 别	示例表达式	结 果
+	二元	var1 = var2 + var3;	var1 的值是 var2 与 var3 的和
-	二元	var1 = var2 - var3;	var1 的值是从 var2 减去 var3 所得的值
*	二元	var1 = var2 * var3;	var1 的值是 var2 与 var3 的乘积
/	二元	var1 = var2 / var3;	var1 是 var2 除以 var3 所得的值
%	二元	var1 = var2 % var3;	var1 是 var2 除以 var3 所得的余数
+	一元	var1 = +var2;	var1 的值等于 var2 的值
-	一元	var1 = - var2;	var1 的值等于 var2 的值乘以 - 1



+(一元)运算符有点古怪,因为它对结果没有影响。它不会把值变成正的:如果 var2 是-1,则+var2 仍是-1。但是,这是一个普遍认可的运算符,所以也把它包含进来。这个运算符最有用的方面是,可以定制它的操作,本书在后面探讨运算符的重载时会介绍它。

上面的示例都使用简单的数值类型,因为使用其他简单类型,结果可能不太清晰。例如把两个布尔值加在一起,会得到什么结果?此时,如果对 bool 变量使用+(或其他数学运算符),编译器会报错。char 变量的相加也会有点让人摸不着头脑。记住, char 变量实际上存储的是数字,所以把两个 char 变量加在一起也会得到一个数字(其类型为 int)。这是一个隐式转换的示例,稍后将详细介绍这个主题和显式转换,因为它也可以应用到 var1、var2 和 var3 都是混合类型的情况。

二元运算符+在用于字符串类型变量时也是有意义的。此时,表 3-7 的表项应如下所示。

表 3-7

运 算 符	类 别	示例表达式	结 果
+	二元	var1 = var2 + var3;	var1 的值是存储在 var2 和 var3 中的两个字符串的连接值

但其他数学运算符不能用于处理字符串。

这里应介绍的另外两个运算符是递增和递减运算符,它们都是一元运算符,可以以两种方式加以使用:放在操作数的前面或后面。简单表达式的结果如表 3-8 所示。

表 3-8

运 算 符	类 别	示例表达式	结 果
++	一元	var1 = ++var2;	var1 的值是 var2 + 1, var2 递增 1
--	一元	var1 = --var2;	var1 的值是 var2 - 1, var2 递减 1
++	一元	var1 = var2++;	var1 的值是 var2, var2 递增 1
--	一元	var1 = var2--;	var1 的值是 var2, var2 递减 1

这些运算符改变存储在操作数中的值。

- ++总是使操作数加 1
- --总是使操作数减 1

var1 中存储的结果有区别,其原因是运算符的位置决定了它什么时候发挥作用。把运算符放在操作数的前面,则操作数是在进行任何其他计算前受到运算符的影响,而把运算符放在操作数的后面,则操作数是在完成表达式的计算后受到运算符的影响。

这有益于另一个示例,考虑以下代码:

```
int var1, var2 = 5, var3 = 6;
var1 = var2++ * --var3;
```

要把什么值赋予 var1? 在计算表达式前, var3 前面的运算符 -- 会起作用,把它的值从 6 改为 5。



可以忽略 var2 后面的++运算符,因为它是在计算完成后才发挥作用,所以 var1 的结果是 5 与 5 的乘积,即 25。

在许多情况下,这些简单的一元运算符使用起来非常方便,它们实际上是下述表达式的简写形式:

```
var1 = var1 + 1;
```

这类表达式有许多用途,特别适合于在循环中使用,这将在第 4 章讲述。下面的示例说明如何使用数学运算符,并介绍另外两个有用的概念。代码提示用户键入一个字符串和两个数字,然后显示计算结果。

### 试一试: 用数学运算符处理变量

- (1) 在目录 C:\BegVCSharp\Chapter03 下创建一个新控制台应用程序 Ch03Ex02。
- (2) 在 Program.cs 中添加如下代码:



```
static void Main(string[] args)
{
    double firstNumber, secondNumber;
    string userName;
    Console.WriteLine("Enter your name:");
    userName = Console.ReadLine();
    Console.WriteLine("Welcome {0}!", userName);
    Console.WriteLine("Now give me a number:");
    firstNumber = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Now give me another number:");
    secondNumber = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
        secondNumber, firstNumber + secondNumber);
    Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
        secondNumber, firstNumber, firstNumber - secondNumber);
    Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
        secondNumber, firstNumber * secondNumber);
    Console.WriteLine("The result of dividing {0} by {1} is {2}.",
        firstNumber, secondNumber, firstNumber / secondNumber);
    Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
        firstNumber, secondNumber, firstNumber % secondNumber);
    Console.ReadKey();
}
```

代码段 Ch03Ex02\Program.cs

- (3) 执行代码,结果如图 3-2 所示。

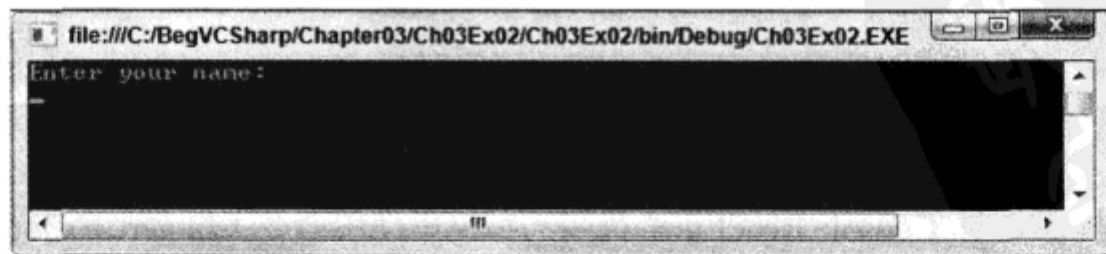


图 3-2

(4) 输入名称，按下回车键，如图 3-3 所示。

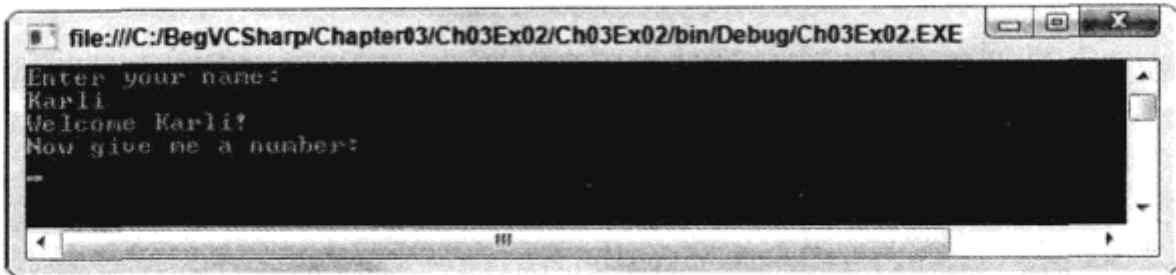


图 3-3

(5) 输入一个数字，按下回车键，再输入另一个数字，按下回车键，如图 3-4 所示。



图 3-4

示例的说明

除了演示数学运算符外，这段代码还引入了两个重要概念，在以后的示例中将多次用到这些概念。

- 用户输入
- 类型转换

用户输入使用与前面 `Console.WriteLine()` 命令类似的语法。但这里使用 `Console.ReadLine()`。这个命令提示用户输入信息，并把它们存储在 `string` 变量中。

```
string userName;  
Console.WriteLine("Enter your name:");  
userName = Console.ReadLine();  
Console.WriteLine("Welcome {0}!", userName);
```

这段代码直接将已赋值变量 `userName` 的内容写到屏幕上。

这个示例还读取了两个数字，下面略微展开讨论一下。因为 `Console.ReadLine()` 命令生成一个字符串，而我们希望得到一个数字，所以这就引入了类型转换的问题。第 5 章将详细讨论类型转换，下面先看看本例使用的代码。

首先，声明要存储数字的变量：

```
double firstNumber, secondNumber;
```

接着，给出提示，对 `Console.ReadLine()` 得到的字符串使用命令 `Convert.ToDouble()`，把字符串转换为 `double` 类型，把这个数值赋给前面声明的变量 `firstNumber`：

```
Console.WriteLine("Now give me a number:");
```

```
firstNumber = Convert.ToDouble(Console.ReadLine());
```

这个语法是相当简单的，其他的许多转换也用类似的方式进行。  
其余的代码按同样的方式获取第二个数：

```
Console.WriteLine("Now give me another number:");
secondNumber = Convert.ToDouble(Console.ReadLine());
```

然后输出两个数字的加、减、乘、除的结果，并使用余数运算符(%)显示除操作的余数。

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
    secondNumber, firstNumber + secondNumber);
Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
    secondNumber, firstNumber, firstNumber - secondNumber);
Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
    secondNumber, firstNumber * secondNumber);
Console.WriteLine("The result of dividing {0} by {1} is {2}.",
    firstNumber, secondNumber, firstNumber / secondNumber);
Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
    firstNumber, secondNumber, firstNumber % secondNumber);
```

注意，我们提供了表达式 `firstNumber + secondNumber` 等，作为 `Console.WriteLine()` 语句的一个参数，而没有使用中间变量：

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
    secondNumber, firstNumber + secondNumber);
```

这种语法可以提高代码的可读性，减少需要编写的代码量。

3.4.2 赋值运算符

我们迄今一直在使用简单的=赋值运算符，其实还有其他赋值运算符，而且它们都很有用。除了=运算符外，其他赋值运算符都以类似的方式工作。与=一样，它们都是根据运算符和右边的操作数，把一个值赋给左边的变量。

表 3-9 列出了这些运算符及其说明。

表 3-9

运 算 符	类 别	示例表达式	结 果
=	二元	var1 = var2;	var1 被赋予 var2 的值
+=	二元	var1 += var2;	var1 被赋予 var1 与 var2 的和
-=	二元	var1 -= var2;	var1 被赋予 var1 与 var2 的差
*=	二元	var1 *= var2;	var1 被赋予 var1 与 var2 的乘积
/=	二元	var1 /= var2;	var1 被赋予 var1 与 var2 相除所得的结果
%=	二元	var1 %= var2;	var1 被赋予 var1 与 var2 相除所得的余数

可以看出，这些运算符把 `var1` 也包括在计算过程中，下面的代码：

```
var1 += var2;
```

与下面的代码结果相同。

```
var1 = var1 + var2;
```



`+=`运算符也可以用于字符串，与`+`运算符一样。

使用这些运算符，特别是在使用长变量名时，可以使代码更便于阅读。

3.4.3 运算符的优先级

在计算表达式时，会按顺序处理每个运算符。但这并不意味着必须从左至右地运用这些运算符。例如，有下面的代码：

```
var1 = var2 + var3;
```

其中`+`运算符就是在`=`运算符之前进行计算的。在其他一些情况下，运算符的优先级并没有这么明显，例如：

```
var1 = var2 + var3 * var4;
```

其中`*`运算符先计算，其后是`+`运算符，最后是`=`运算符，这是标准的数学运算顺序，其结果与我们在纸上进行算术运算的结果相同。

像这样的计算，可以使用括号控制运算符的优先级，例如：

```
var1 = (var2 + var3) * var4;
```

先计算括号中的内容，即`+`运算符在`*`运算符之前计算。

对于前面介绍的运算符，其优先级如表 3-10 所示，优先级相同的运算符(如`*`和`/`)按照从左至右的顺序计算。

表 3-10

优 先 级	运 算 符
优先级由高到低	<code>++</code> , <code>--</code> (用作前缀); <code>+</code> , <code>-</code> (一元)
	<code>*</code> , <code>/</code> , <code>%</code>
	<code>+</code> , <code>-</code>
	<code>=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>+=</code> , <code>-=</code>
	<code>++</code> , <code>--</code> (用作后缀)



如上所述，括号可用于重写优先级顺序。另外，`++`和`--`用作后缀运算符时，在概念上其优先级最低，如上表所示。它们不对赋值表达式的结果产生影响，所以可以认为它们的优先级比所有其他运算符都高。但是，它们会在计算表达式后改变操作数的值，所以很容易认可它们在上表中的优先级。

### 3.4.4 名称空间

在继续学习前,应花一定的时间了解一个比较重要的主题——名称空间。它们是.NET 中提供应用程序代码容器的方式,这样就可以唯一地标识代码及其内容。名称空间也用作.NET Framework 中给项分类的一种方式。大多数项都是类型定义,例如,本章描述的简单类型(System.Int32 等)。

默认情况下,C#代码包含在全局名称空间中。这意味着对于包含在这段代码中的项,只要按照名称进行引用,就可以由全局名称空间中的其他代码访问它们。可以使用 `namespace` 关键字为花括号中的代码块显式定义名称空间。如果在该名称空间代码的外部使用名称空间中的名称,就必须写出该名称空间中的限定名称。

限定名称包括它所有的分层信息。这基本上意味着,如果一个名称空间中的代码需要使用在另一个名称空间中定义的名称,就必须包括对该名称空间的引用。限定名称在不同的命名空间级别之间使用句点字符(.)。如下所示:

```
namespace LevelOne
{
    // code in LevelOne namespace

    // name "NameOne" defined
}

// code in global namespace
```

这段代码定义了一个名称空间 `LevelOne`,以及该名称空间中的一个名称 `NameOne`(注意这里没有列出其他代码,是为了使我们的讨论更具普遍性,并在定义名称空间的地方添加了一个注释)。在名称空间 `LevelOne` 中编写的代码可以使用 `NameOne` 来引用该名称,不需要任何分类信息。但全局名称空间中的代码必须使用分类名称 `LevelOne.NameOne` 来引用这个名称。



根据约定,名称空间通常采用 `PascalCase` 命名方式。

在名称空间中,使用关键字 `namespace` 还可以定义嵌套的名称空间。嵌套的名称空间通过其层次结构来引用,并使用句点区分层次结构的层次。这最好通过一个示例来加以说明。考虑下面的名称空间:

```
namespace LevelOne
{
    // code in LevelOne namespace

    namespace LevelTwo
    {
        // code in LevelOne.LevelTwo namespace

        // name "NameTwo" defined
    }
}

// code in global namespace
```



在全局名称空间中, NameTwo 必须引用为 LevelOne.LevelTwo.NameTwo; 在 LevelOne 名称空间中, 可以引用为 LevelTwo.NameTwo; 在 LevelOne.LevelTwo 名称空间中, 可以引用为 NameTwo。

非常重要的一点是, 名称是由名称空间唯一定义的。可以在 LevelOne 和 LevelTwo 名称空间中定义名称 NameThree:

```
namespace LevelOne
{
    // name "NameThree" defined

    namespace LevelTwo
    {
        // name "NameThree" defined
    }
}
```

这定义了两个不同的名称 LevelOne.NameThree 和 LevelOne.LevelTwo.NameThree, 可以独立使用它们, 互不干扰。

创建了名称空间后, 即可使用 using 语句简化对它们所含名称的访问。实际上, using 语句的意思是“我们需要这个名称空间中的名称, 所以不要每次总是要求对它们分类”。例如, 在下面的代码中, LevelOne 名称空间中的代码可以访问 LevelOne.LevelTwo 名称空间中的名称, 而无需分类:

```
namespace LevelOne
{
    using LevelTwo;

    namespace LevelTwo
    {
        // name "NameTwo" defined
    }
}
```

LevelOne 命名空间中的代码现在可以直接使用 NameTwo 引用 LevelTwo.NameTwo。

有时, 与上面的 NameThree 示例一样, 不同名称空间中的相同名称会产生冲突, 使系统崩溃(此时, 代码无法编译, 编译器会告诉我们名称有冲突)。此时, 可以使用 using 语句为名称空间提供一个别名。

```
namespace LevelOne
{
    using LT = LevelTwo;

    // name "NameThree" defined

    namespace LevelTwo
    {
        // name "NameThree" defined
    }
}
```

LevelOne 名称空间中的代码可以把 LevelOne.NameThree 引用为 NameThree, 把 LevelOne.LevelTwo.NameThree 引用为 LT.NameThree。

using 语句可以应用到包含它们的名称空间，以及该名称空间中包含的嵌套命名空间。在上面的代码中，全局名称空间不能使用 LT.NameThree。但如果 using 语句声明如下：

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
    // name "NameThree" defined

    namespace LevelTwo
    {
        // name "NameThree" defined
    }
}
```

全局名称空间中的代码和 LevelOne 名称空间中的代码就可以使用 LT.NameThree。

需要注意特别重要的一点：using 语句本身不能访问另一个名称空间中的名称。除非名称空间中的代码以某种方式链接到项目上，或者代码是在该项目的源文件中定义的，或在链接到该项目的其他代码中定义的，否则就不能访问其中包含的名称。另外，如果包含名称空间的代码链接到项目上，无论是否使用 using，都可以访问其中包含的名称。using 语句便于我们访问这些名称，减少代码量，以及提高可读性。

回过头来看看本章开头的 ConsoleApplication1 中的代码，下面的代码被应用到名称空间上：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    ...
}
```

以 using 关键字开头的 4 行代码声明在这段 C# 代码中使用 System、System.Collections.Generic、System.Linq 和 System.Text 名称空间，它们可以在该文件的所有名称空间中访问，无需分类。System 名称空间是 .NET Framework 应用程序的根命名空间，包含控制台应用程序需要的所有基本功能。其他 3 个名称空间常用于控制台应用程序，所以该程序包含了这 4 行代码。

最后，为应用程序代码本身声明一个名称空间 ConsoleApplication1。

### 3.5 小结

本章介绍了创建有效 C# 应用程序的大量基础知识，讲述了 C# 的基本语法，分析了在创建控制台应用程序项目时 VS 和 VCE 生成的基本控制台应用程序代码。

本章重点讲述变量的用法。我们描述了变量的含义，阐述了如何创建变量，如何给它们赋值，如何处理它们以及它们包含的值。同时，介绍了一些基本的用户交互，描述了如何把文本输出到控制台应用程序上，如何读取用户的输入。这涉及到一些非常基本的类型转换。类型转换是一个复杂

的主题，将在第 5 章详细论述。

本章还介绍了如何将运算符和操作数组合为表达式，并说明了这些运算符的执行方式，以及它们的执行顺序。

最后介绍了名称空间。随着本书内容的深入，命名空间会显得越来越重要。这里仅以比较抽象的方式介绍了这个主题，完整的论述请见本书后面的内容。

到目前为止，所有的编程工作都是逐行完成的。第 4 章将学习如何使用循环技术和条件分支控制程序执行的流程，以便提高代码的效率。

## 3.6 练习

(1) 在下面的代码中，如何从名称空间 `fabulous` 的代码中引用名称 `great`?

```
namespace fabulous
{
    // code in fabulous namespace
}

namespace super
{
    namespace smashing
    {
        // great name defined
    }
}
```

(2) 下面哪些变量名不合法?

- `myVariableIsGood`
- `99Flake`
- `_floor`
- `time2GetJiggyWidIt`
- `wrox.com`

(3) 字符串 `supercalifragilisticexpialidocious` 是因为太长了而不能放在 `string` 变量中吗? 为什么?

(4) 考虑运算符的优先级，列出下述表达式的计算步骤。

```
resultVar += var1 * var2 + var3 % var4 / var5;
```

(5) 编写一个控制台应用程序，要求用户输入 4 个 `int` 值，并显示它们的乘积。提示：可以使用 `Convert.ToDouble()` 命令，把用户在控制台上输入的数转换为 `double`；从 `string` 转换为 `int` 的命令是 `Convert.ToInt32()`。

附录 A 给出了练习答案。

3.7 本章要点

主 题	重 要 概 念
C#基本语法	C#是一种区分大小写的语言，每行代码都以分号结束。如果代码行太长或者表示嵌套的块，可以缩进代码行，以方便阅读。使用//或/*...*/语法可以包含不编译的注释。代码块可以隐藏到区域中，也是为了方便阅读
变量	变量是有名称和类型的数据块。.NET Framework 定义了大量的简单类型，例如数字和字符串(文本)类型，以供使用。变量只有经过声明和初始化后，才能使用。可以把字面值赋予变量，以初始化它们，变量还可以在单个步骤中声明和初始化
表达式	表达式利用运算符和操作数来建立，其中运算符对操作数执行操作。运算符有 3 种：一元、二元和三元运算符，它们分别操作 1、2 和 3 个操作数。数学运算符对数值执行操作，赋值运算符把表达式的结果放在变量中。运算符有固定的优先级，优先级确定了运算符在表达式中的处理顺序
名称空间	.NET 应用程序中定义的所有名称，包括变量名，都包含在名称空间中。名称空间采用层次结构，我们通常需要根据包含名称的名称空间来限定名称，以便访问它们







# 第 4 章

## 流程控制

### 本章内容:

- 布尔逻辑的含义及其用法
- 如何控制代码的分支
- 如何编写循环代码

我们迄今看到的 C# 代码有一个共同点：程序的执行都是一行接一行、自上而下地进行，不遗漏任何代码。如果所有应用程序都这样执行，则我们能做的工作就很有限了。本章介绍控制程序流的两种方法。程序流程就是 C# 代码的执行顺序。这两种方法是分支和循环。分支是有条件地执行代码。条件取决于计算的结果，例如，“只有 `myVal` 小于 10，才执行这行代码”。循环重复执行相同的语句（重复执行一定的次数，或者在满足测试条件后停止执行）。

这两种方法都要用到布尔逻辑。第 3 章介绍了 `bool` 类型，但并未讨论它。本章将在很多地方使用它，所以先讨论布尔逻辑，以便在流程控制环境下使用它。

### 4.1 布尔逻辑

第 3 章介绍的 `bool` 类型可以有两个值：`true` 或 `false`。这种类型常常用于记录某些操作的结果，以便操作这些结果。`bool` 类型可用于存储比较结果。



19 世纪中叶的英国数学家乔治·布尔为布尔逻辑奠定了基础。

考虑下述情形(如本章引言所述)：要根据变量 `myVal` 是否小于 10，来确定是否执行代码。为此，需要确定语句“`myVal` 小于 10”的真假，即需要了解比较的布尔结果。

布尔比较需要使用布尔比较运算符(也称为关系运算符), 如表 4-1 所示。这里 var1 都是 bool 类型的变量, var2 和 var3 则可以是不同类型。

表 4-1

运算符	类别	示例表达式	结果
=	二元	var1 = var2 == var3;	如果 var2 等于 var3, var1 的值就是 true, 否则为 false
!=	二元	var1 = var2 != var3;	如果 var2 不等于 var3, var1 的值就是 true, 否则为 false
<	二元	var1 = var2 < var3;	如果 var2 小于 var3, var1 的值就是 true, 否则为 false
>	二元	var1 = var2 > var3;	如果 var2 大于 var3, var1 的值就是 true, 否则为 false
<=	二元	var1 = var2 <= var3;	如果 var2 小于等于 var3, var1 的值就是 true, 否则为 false
>=	二元	var1 = var2 >= var3;	如果 var2 大于等于 var3, var1 的值就是 true, 否则为 false

在代码中, 可以对数值使用以下这些运算符:

```
bool isLessThan10;  
isLessThan10 = myVal < 10;
```


如果 myVal 存储的值小于 10, 这段代码就给 isLessThan10 赋予 true 值, 否则赋予 false 值。也可以对其他类型使用这些比较运算符, 例如字符串:

```
bool isKarli;  
isKarli = myString == "Karli";
```

如果 myString 存储的字符串是 Karli, isKarli 的值就为 true。也可以对布尔值使用这些运算符:

```
bool isTrue;  
isTrue = myBool == true;
```

但只能使用==和!=运算符。



一个常见的代码错误是, 无意间假定由于 val1 < val2 是 false, 所以 val1 > val2 为 true。如果 val1 == val2, 则这两个语句都是 false。

在处理布尔值时, 还有其他一些布尔运算符, 如表 4-2 所示。

表 4-2

运算符	类别	示例表达式	结果
!	一元	var1 = ! var2;	如果 var2 是 false, var1 的值就是 true, 否则为 false(逻辑非)
&	二元	var1 = var2 & var3;	如果 var2 和 var3 都是 true, var1 的值就是 true, 否则为 false(逻辑与)
	二元	var1 = var2   var3;	如果 var2 或 var3 是 true(或两者都是), var1 的值就是 true, 否则为 false(逻辑或)
^	二元	var1 = var2 ^ var3;	如果 var2 或 var3 中有且仅有一个是 true, var1 的值就是 true, 否则为 false (逻辑异或)

上面的代码也可以表述为:

```
bool isTrue;  
isTrue = myBool & true;
```

&和 | 运算符也有两个类似的运算符, 称为条件布尔运算符(见表 4-3)。

表 4-3

运算符	类别	示例表达式	结果
&&	二元	var1 = var2 && var3;	如果 var2 和 var3 都是 true, var1 的值就是 true, 否则为 false (逻辑与)
	二元	var1 = var2    var3;	如果 var2 或 var3 是 true(或两者都是), var1 的值就是 true, 否则为 false (逻辑或)

这些运算符的结果与&和 | 完全相同, 但得到结果的方式有一个重要区别: 其性能比较好。两者都是检查第一个操作数的值(表 4-3 中的 var2), 再根据该操作数的值进行操作, 可能根本就不处理第二个操作数(表 4-3 中的 var3)。

如果&&运算符的第一个操作数是 false, 就不需要考虑第二个操作数的值了, 因为无论第二个操作数的值是什么, 其结果都是 false。同样, 如果第一个操作数是 true, || 运算符就返回 true, 无需考虑第二个操作数的值。但上面的&和 | 运算符却不是这样。它们总是要计算两个操作数。

因为操作数的计算是有条件的, 如果使用&&和||运算符来代替&和 |, 性能会有一定提高。在大量使用这些运算符的应用程序中这表现得尤为明显。作为一个规则, 尽可能使用&&和 || 运算符。这些运算符有时用于比较复杂的情形, 例如, 只有第一个操作数包含某个值时, 才计算第二个操作数:

```
var1 = (var2 != 0) && (var3 / var2 >2);
```

如果 var2 是 0, 则 var3 除以 var2 就会导致“除 0 错误”, 或者把 var1 定义为无穷大(对于某些类型如 float 来说, 可能出现后一种情形, 也是可以检测到的)。



读者此时可能会问，为什么会有&和|运算符。原因是这两个运算符可以用于对数值执行操作。实际上，它们处理的是存储在变量中的一系列位，而不是变量的值。请参见稍后的“按位运算符”。

4.1.1 布尔赋值运算符

使用布尔赋值运算符可以把布尔比较与赋值组合起来，其方式与第3章中的数学赋值运算符(+=, \*=等)相同。布尔值如表 4-4 所示。

表 4-4

运算符	类别	示例表达式	结果
&=	二元	var1 &= var2;	var1 的值是 var1 & var2 的结果
=	二元	var1  = var2;	var1 的值是 var1   var2 的结果
^=	二元	var1 ^= var2;	var1 的值是 var1 ^ var2 的结果

这些运算符处理布尔值和数值的方式与&、| 和 ^ 相同。



&=和|=赋值运算符并不使用&&和||条件布尔运算符，即无论赋值运算符左边的值是什么，都处理所有的操作数。

在下面的示例中，用户键入一个整数，然后代码使用该整数执行各种布尔运算。

试一试：使用布尔运算符

- (1) 在目录 C:\BegVCSharp\Chapter04 下创建一个新控制台应用程序 Ch04Ex01。
- (2) 把以下代码添加到 Program.cs 中：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    Console.WriteLine("Enter an integer:");
    int myInt = Convert.ToInt32(Console.ReadLine());
    bool isLessThan10 = myInt < 10;
    bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
    Console.WriteLine("Integer less than 10? {0}", isLessThan10);
    Console.WriteLine("Integer between 0 and 5? {0}", isBetween0And5);
    Console.WriteLine("Exactly one of the above is true? {0}",
        isLessThan10 ^ isBetween0And5);
    Console.ReadKey();
}
```

代码段 Ch04Ex01\Program.cs

- (3) 运行应用程序，出现提示时，输入一个整数，结果如图 4-1 所示。

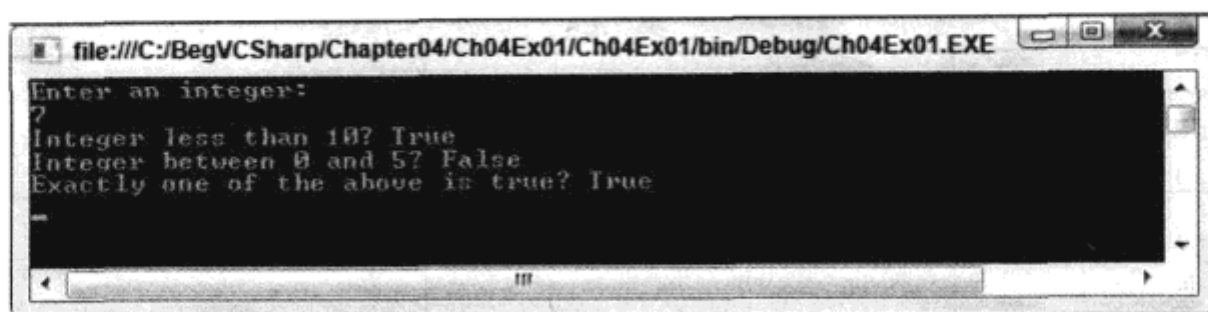


图 4-1

### 示例的说明

前两行代码使用前面介绍的技术，提示并接受一个整数值：

```
Console.WriteLine("Enter an integer:");
int myInt = Convert.ToInt32(Console.ReadLine());
```

使用 `Convert.ToInt32()` 从字符串输入中得到一个整数。`Convert.ToInt32()` 是另一个类型转换命令，与前面使用的 `Convert.ToDouble()` 命令属于同一系列。

接着声明两个布尔变量 `isLessThan10` 和 `isBetween0And5`，并赋值，其中的逻辑匹配其名称中的描述：

```
bool isLessThan10 = myInt < 10;
bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
```

接着在下面的 3 行代码中使用这些变量，前两行代码输出它们的值，第 3 行对它们执行一个操作，并输出结果。在执行这段代码时，假定用户输入了 7，如图 4-1 所示。

第一个输出是操作 `myInt < 10` 的结果。如果 `myInt` 是 6，则它小于 10，因此结果为 `true`。如果 `MyInt` 的值是 10 或更大，就会得到 `false`。

第二个输出涉及较多计算：`(0 <= myInt) && (myInt <= 5)`，其中包含两个比较操作，用于确定 `myInt` 是否大于或等于 0，且小于或等于 5。接着对结果进行布尔 AND 操作。输入数字 6，则 `(0 <= myInt)` 返回 `true`，而 `(myInt <= 5)` 返回 `false`，最终结果就是 `(true) && (false)`，即 `false`，如图 4-1 所示。

最后，对两个布尔变量 `isLessThan10` 和 `isBetween0And5` 执行逻辑异或操作。如果一个变量的值是 `true`，另一个是 `false`，则代码返回 `true`，所以只有 `myInt` 是 6、7、8 或 9，才返回 `true`，本例输入的是 6，所以结果是 `true`。

#### 4.1.2 按位运算符

前面介绍的 `&` 和 `|` 运算符还有一个作用：对数值执行操作。以这种方式使用时，它们处理的是变量中存储的一系列位，而不是变量值，因此它们称为按位运算符。

本节介绍它们和 C# 语言定义的其他按位运算符。在大多数开发工作中，除了数学应用之外，这个功能都不太常用。因此本节没有示例。

下面先讨论 `&` 和 `|`。第一个操作数中的每个位都与第二个操作数中相同位置上的位进行比较，在得到的结果中，各个位置上的位如表 4-5 所示。

`|` 运算符与此类似，但得到的结果位是不同的，如表 4-6 所示。



表 4-5

操作数 1 的位	操作数 2 的位	&的结果位
1	1	1
1	0	0
0	1	0
0	0	0

表 4-6

操作数 1 的位	操作数 2 的位	的结果位
1	1	1
1	0	1
0	1	1
0	0	0

例如，考虑下面代码中的操作：

```
int result, op1, op2;
op1 = 4;
op2 = 5;
result = op1 & op2;
```

这里必须考虑 op1 和 op2 的二进制表示方式，它们分别是 100 和 101。比较这两个表达方式中相同位置上的二进制数字，得出结果，如下所示：

- 如果 op1 和 op2 最左边的位都是 1，result 最左边的位就是 1，否则为 0。
- 如果 op1 和 op2 次左边的位都是 1，result 次左边的位就是 1，否则为 0。
- 继续比较其他的位。

在这个示例中，op1 和 op2 最左边的位都是 1，所以 result 最左边的位就是 1。下一个位都是 0，第 3 个位置上的位分别是 1 和 0，则 result 第 2~3 个位都是 0。最后，结果的二进制值是 100，即结果是 4。以下是这个过程：

100

&

101

100

4

100

&

101

100

4

如果使用 | 运算符，将进行相同的过程，但如果操作数中相同位置上的位有一个是 1，其结果位就是 1，如下所示：

100

4

1101

5

101

5

^运算符的用法与此相同。如果操作数中相同位置上的位有且仅有一个是1，其结果位就是1，如表 4-7 所示。

表 4-7

操作数 1 的位	操作数 2 的位	^的结果位
1	1	0
1	0	1
0	1	1
0	0	0

C#中还可以使用一元位运算符~，它将操作数中的位取反，其结果应是操作数中位为1的，在结果中就是0，反之亦然，如表 4-8 所示。

表 4-8

操作数的位	~的结果位
1	0
0	1

整数存储在.NET 中的方式称为2的补位，即使用一元运算符~会使结果看起来有点古怪。假定int 类型是一个32位的数字，则运算符~对所有32位进行操作，将有助于看出这种方式。例如，数字5的完整二进制表示为：

00000000000000000000000000000101

数字-5的完整二进制表示为：

1111111111111111111111111111011

实际上，按照2的补位系统，(-x)定义为(~x+1)。这个系统在把数字加在一起时非常有用。例如，把10和-5加起来(即从10中减去5)的二进制表示为：

00000000000000000000000000001010  
+ 1111111111111111111111111111011  
= 1000000000000000000000000000101



忽略最左端的1，就得到5的二进制表示。像~1=2这样的式子比较古怪，其原因是底层的结构强制生成了这个结果。

在某些情况下，本节介绍的这些位运算符是非常有用的，因为它们可以用变量中的各个位存储信息。例如，颜色可以使用 3 个位来指定红、绿、蓝。可以分别设置这些位，改变这 3 个位，进行以下一种配置，如表 4-9 所示。

表 4-9

位	十 进 制 数	含 义
000	0	黑色
100	4	红色
010	2	绿色
001	1	蓝色
101	5	洋红色
110	6	黄色
011	3	青色
111	7	白色

假定把这些值存储在一个类型为 `int` 的变量中。首先从黑色开始，即值为 0 的 `int` 变量，可以执行如下操作：

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;           // Add green bit, myColor now stores 010
myColor = myColor | 4;           // Add red bit, myColor now stores 110
containsRed = (myColor & 4) == 4; // Check value of red bit
```

最后一行代码将值 `true` 赋予 `containsRed`，因为 `myColor` 的“红色位”是 1。这种技术在高效使用信息时非常有效，特别适合于同时检查多个位的值(对于 `int` 值，是 32 位)。但是，在单个变量中存储额外信息有更好的方式，即利用第 5 章讨论的高级变量类型。

除了这 4 个按位运算符外，本节还要介绍另外两个运算符，如表 4-10 所示。

表 4-10

运 算 符	类 别	示例表达式	结 果
>>	二元	<code>var1 = var2 &gt;&gt; var3;</code>	把 <code>var2</code> 的二进制值向右移动 <code>var3</code> 位，就得到 <code>var1</code> 的值
<<	二元	<code>var1 = var2 &lt;&lt; var3;</code>	把 <code>var2</code> 的二进制值向左移动 <code>var3</code> 位，就得到 <code>var1</code> 的值

这些运算符通常称为位移运算符，最好用一个简单的示例加以说明：

```
int var1, var2 = 10, var3 = 2;
var1 = var2 << var3;
```

结果，`var1` 的值是 40。具体过程如下：10 的二进制值是 1010，把该数值向左移动两位，得到 101000，即十进制中的 40。实际上，是执行了乘法操作。每向左移动一位，该数都要乘以 2，所以向左移动两位，就给原来的操作数乘以 4。而每向右移动一位，则是给操作数除以 2，并丢弃非整数余数：

```
int var1, var2 = 10;
var1 = var2 >> 1;
```

在这个示例中，var1 的值是 5，而下面的代码得到的值是 2:

```
int var1, var2 = 10;
var1 = var2 >> 2;
```

在大多数代码中，都不使用这些运算符，但应知道有这样的运算符存在。它们主要用于高度优化的代码，在这些代码中，不能使用其他数学操作。因此它们通常用于设备驱动程序或系统代码。

位移运算符也有赋值运算符，如表 4-11 所示。

表 4-11

运 算 符	类 别	示例表达式	结 果
>>=	一元	var1 >>= var2;	把 var1 的二进制值向右移动 var2 位，就得到 var1 的值
<<=	一元	var1 <<= var2;	把 var1 的二进制值向左移动 var2 位，就得到 var1 的值

4.1.3 运算符优先级的更新

现在要考虑更多的运算符，所以应更新第 3 章中的运算符优先级表，把它们包括在内，如表 4-12 所示。

表 4-12

优 先 级	运 算 符
优 先 级 由 高 到 低	++, -(用作前缀); 0, +, - (一元), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	&&
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=,  =
	++, --(用作后缀)

该表增加了好几个级别，但它明确定义了下述表达式该如何计算:

```
var1 = var2 <= 4 && var2 >= 2;
```

其中&&运算符在<= 和 >=运算符之后执行(在这行代码中，var2 是一个 int 值)。

这里要注意的是，添加括号可以使这样的表达式看起来更清晰。编译器知道用什么顺序执行运算符，但人们常常会忘记这个顺序(有时可能想改变这个顺序)。上述表达式也可以写为：

```
var1 = (var2 <= 4) && (var2 >= 2);
```

要解决这个问题，可以明确指定计算的顺序。

## 4.2 goto 语句

C#允许给代码行加上标签，这样就可以使用 `goto` 语句直接跳转到这些代码行上。该语句优缺点并存。主要的优点是：这是控制什么时候执行哪些代码的一种简单方式。主要的缺点是：过多地使用这个技巧将使代码晦涩难懂。

`goto` 语句的用法如下：

```
goto <labelName>;
```

标签用下述方式定义：

```
<labelName>:
```

例如，下面的代码：

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

其执行过程如下：

- `myInteger` 声明为 `int` 类型，并赋予值 5。
- `goto` 语句中断正常的执行过程，把控制权转到标有 `myLabel:` 的代码行上。
- `myInteger` 的值写入控制台。

下面的第 3 行代码从未执行。

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

实际上，如果在应用程序中加入这段代码，会发现编译代码时，`Error List` 窗口会显示一个警告，即 `Unreachable code detected` 和一个行号。在无法执行的代码行中，`myInteger` 下面还有绿色的波浪线。

`goto` 语句有它们的作用，但也可能使代码陷入混乱。尽量不要使用它(使用本章后面介绍的技巧，就可以避免使用它)。例如，因使用 `goto` 语句而非常难懂的代码如下所示：

```
start:
int myInteger = 5;
```



```

goto addVal;
writeResult:
Console.WriteLine("myInteger = {0}", myInteger);
goto start;
addVal:
myInteger += 10;
goto writeResult;

```

这是有效的代码，但非常难懂，读者可以自己试试，看看会发生什么情况。在此之前，应尝试理解这些代码会完成什么任务。后面再分析这个语句，因为本章的其他一些结构将使用该语句。

## 4.3 分支

分支是控制下一步要执行哪行代码的过程。要跳转到的代码行由某个条件语句来控制。这个条件语句使用布尔逻辑，对测试值和一个或多个可能的值进行比较。

本节介绍 C# 中的 3 种分支技术：

- 三元运算符
- if 语句
- switch 语句

### 4.3.1 三元运算符

最简单的比较方式是使用第 3 章介绍的三元(或条件)运算符。一元运算符有一个操作数，二元运算符有两个操作数，所以三元运算符有 3 个操作数。其语法如下：

```
<test> ? <resultIfTrue> : <resultIfFalse>
```

其中，计算<test> 可得到一个布尔值，运算符的结果根据这个值来确定是<resultIfTrue>，还是<resultIfFalse>。

使用三元运算符可以测试 int 变量 myInteger 的值：

```

string resultString = (myInteger < 10) ? "Less than 10"
                        : "Greater than or equal to 10";

```

三元运算符的结果是两个字符串中的一个，这两个字符串都可能赋给 resultString。把哪个字符串赋给 resultString，取决于 myInteger 的值与 10 的比较。如果 myInteger 的值小于 10，就把第一个字符串赋给 resultString；如果 myInteger 的值大于或等于 10，就把第二个字符串赋给 resultString。例如，如果 myInteger 的值是 4，则 resultString 的值就是字符串"Less than 10"。

这个运算符比较适用于这样的简单赋值语句，但不适用于根据比较结果执行大量代码的情形。此时应使用 if 语句。

### 4.3.2 if 语句

if 语句的功能比较多，是有效的决策方式。与?:语句不同的是，if 语句没有结果(所以不在赋值语句中使用它)，使用该语句是为了有条件地执行其他语句。

if 语句最简单的语法如下：

```
if (<test>)
    <code executed if <test> is true>;
```

先执行<test>(其计算结果必须是一个布尔值, 这样代码才能编译), 如果<test>的计算结果是 true, 就执行该语句之后的代码。在这段代码执行完毕后, 或者因为<test>的计算结果是 false, 而没有执行这段代码, 将继续执行后面的代码行。

也可以将 else 语句和 if 语句合并使用, 指定其他代码。如果<test>的计算结果是 false, 就执行 else 语句:

```
if (<test>)
    <code executed if <test> is true>;
else
    <code executed if <test> is false>;
```

可以使用成对的花括号将这两段代码放在多个代码行上:

```
if (<test>)
{
    <code executed if <test> is true>;
}
else
{
    <code executed if <test> is false>;
}
```

例如, 重新编写上一节使用三元运算符的代码:

```
string resultString = (myInteger < 10) ? "Less than 10"
                                : "Greater than or equal to 10";
```

因为 if 语句的结果不能赋给一个变量, 所以要单独将值赋给变量:

```
string resultString;
if (myInteger < 10)
    resultString = "Less than 10";
else
    resultString = "Greater than or equal to 10";
```

这样的代码尽管比较冗长, 但与三元运算符相比, 更便于阅读和理解, 也更加灵活。下面的示例演示了 if 语句的用法。

### 试一试: 使用 if 语句

- (1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新控制台应用程序 Ch04Ex02。
- (2) 把下列代码添加到 Program.cs 中:



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string comparison;
    Console.WriteLine("Enter a number:");
    double var1 = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter another number:");
    double var2 = Convert.ToDouble(Console.ReadLine());
```

```

    if (var1 < var2)
        comparison = "less than";
    else
    {
        if (var1 == var2)
            comparison = "equal to";
        else
            comparison = "greater than";
    }
    Console.WriteLine("The first number is {0} the second number.",
                      comparison);
    Console.ReadKey();
}

```

代码段 Ch04Ex02\Program.cs

(3) 执行代码，根据提示输入两个数字，如图 4-2 所示。

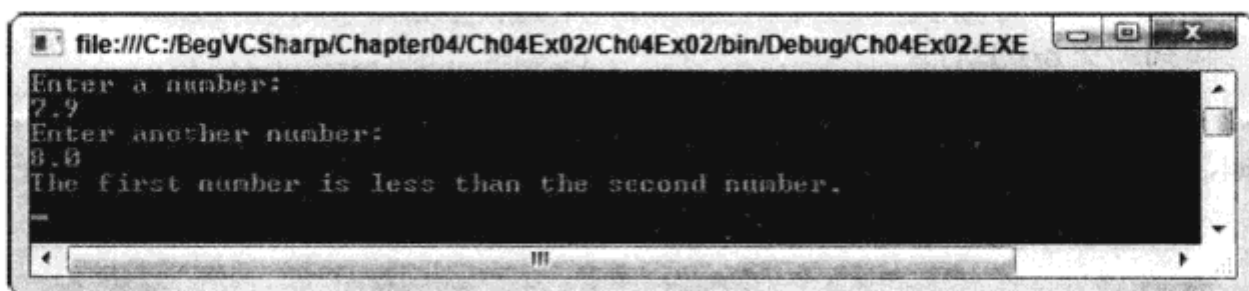


图 4-2

#### 示例的说明

我们已经很熟悉了代码的第一部分，它从用户输入中得到两个 double 值：

```

string comparison;
Console.WriteLine("Enter a number:");
double var1 = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Enter another number:");
double var2 = Convert.ToDouble(Console.ReadLine());

```

接着根据 var1 和 var2 的值，把一个字符串赋给 string 变量 comparison。首先看看 var1 是否小于 var2：

```

if (var1 < var2)
    comparison = "less than";

```

如果不是，则 var1 大于或等于 var2。在第一个比较操作的 else 部分，需要嵌套第二个比较：

```

else
{
    if (var1 == var2)
        comparison = "equal to";
}

```

只有在 var1 大于 var2 时，才执行第二个比较操作中的 else 部分：

```

else
    comparison = "greater than";
}

```

最后将比较操作的值写到控制台上：

```
Console.WriteLine("The first number is {0} the second number.",  
comparison);
```

这里使用的嵌套只是进行这些比较的一种方式，还可以编写如下代码：

```
if (var1 < var2)  
    comparison = "less than";  
if (var1 == var2)  
    comparison = "equal to";  
if (var1 > var2)  
    comparison = "greater than";
```

这个方式的缺点在于无论 `var1` 和 `var2` 的值是什么，都要执行 3 个比较操作。在第一种方式中，如果 `var1 < var2` 是 `true`，就只执行一个比较，否则就要执行两个比较操作(还执行了 `var1 == var2` 比较操作)，这样将使执行的代码行较少。其性能上的差异比较小，但在较重视速度的应用程序中，性能的差异就很明显了。

### 使用 if 语句判断更多的条件

在上面的示例中，有 3 个条件涉及到 `var1` 的值，包括了这个变量所有可能的值。有时要检查特定的值，例如，`var1` 是否等于 1、2、3 或 4 等。使用上面那样的代码会得到很多烦人的嵌套代码：

```
if (var1 == 1)  
{  
    // Do something.  
}  
else  
{  
    if (var1 == 2)  
    {  
        // Do something else.  
    }  
    else  
    {  
        if (var1 == 3 || var1 == 4)  
        {  
            // Do something else.  
        }  
        else  
        {  
            // Do something else.  
        }  
    }  
}
```



**常见错误：**常会错误地将诸如 `if(var1==3 || var1==4)` 的条件写为 `if(var1==3 || 4)`。由于运算符有优先级，因此先执行 `==` 运算符，接着用 `||` 运算符处理布尔和数值操作数，就会出现错误。

在这些情况下，就要使用稍有不同的缩进模式，缩短 `else` 代码块(即在 `else` 块的后面使用一行代码，而不是一个代码块)，这样就得到 `else if` 语句结构。

```

if (var1 == 1)
{
    // Do something.
}
else if (var1 == 2)
{
    // Do something else.
}
else if (var1 == 3 || var1 == 4)
{
    // Do something else.
}
else
{
    // Do something else.
}

```

这些 `else if` 语句实际上是两个独立语句，它们的功能与上述代码相同。但更便于阅读。像这样进行多个比较的操作，应考虑使用另一种分支结构：`switch` 语句。

### 4.3.3 switch 语句

`switch` 语句非常类似于 `if` 语句，因为它也是根据测试的值来有条件地执行代码。但是，`switch` 语句可以一次将测试变量与多个值进行比较，而不是仅测试一个条件。这种测试仅限于离散的值，而不是像“大于 X”这样的子句，所以它的用法有点不同，但它仍是一种强大的技术。

`switch` 语句的基本结构如下：

```

switch (<testVar>)
{
    case <comparisonVal1>:
        <code to execute if <testVar> == <comparisonVal1> >
        break;
    case <comparisonVal2>:
        <code to execute if <testVar> == <comparisonVal2> >
        break;
    ...
    case <comparisonValN>:
        <code to execute if <testVar> == <comparisonValN> >
        break;
    default:
        <code to execute if <testVar> != comparisonVals>
        break;
}

```

`<testVar>` 中的值与每个 `<comparisonValX>` 值(在 `case` 语句中指定)进行比较，如果有一个匹配，就执行为该匹配提供的语句。如果没有匹配，就执行 `default` 部分中的代码。

执行完每个部分中的代码后，还需有另一个语句 `break`。在执行完一个 `case` 块后，再执行第二个 `case` 语句是非法的。



在此，C#与C++是有区别的，在C++中，可以在运行完一个 `case` 语句后，运行另一个 `case` 语句。

这里的 `break` 语句将中断 `switch` 语句的执行，而执行该结构后面的语句。

在 C# 代码中，还有一种方法可以防止程序流程从一个 `case` 语句转到下一个 `case` 语句。即使用 `return` 语句，中断当前函数的运行，而不是仅中断 `switch` 结构的执行(详见第 6 章)。也可以使用 `goto` 语句(如前所述)，因为 `case` 语句实际上是在 C# 代码中定义的标签。例如：

```
switch (<testVar>)
{
    case <comparisonVal1>:
        <code to execute if <testVar> == <comparisonVal1> >
        goto case <comparisonVal2>;
    case <comparisonVal2>:
        <code to execute if <testVar> == <comparisonVal2> >
        break;
    ...
}
```

一个 `case` 语句处理完后，不能自由进入下一个 `case` 语句，但这个规则有一个例外。如果把多个 `case` 语句放在一起(堆叠它们)，其后加一个代码块，实际上是一次检查多个条件。如果满足这些条件中的任何一个，就会执行代码，例如：

```
switch (<testVar>)
{
    case <comparisonVal1>:
    case <comparisonVal2>:
        <code to execute if <testVar> == <comparisonVal1> or
        <testVar> == <comparisonVal2> >
        break;
    ...
}
```

注意，这些条件也应用到 `default` 语句。`default` 语句不一定要放在比较操作列表的最后，还可以把它和 `case` 语句放在一起。用 `break`、`goto` 或 `return` 添加一个断点，可以确保在任何情况下，该结构都有一个有效的执行路径。

每个 `<comparisonValX>` 都必须是一个常数值。一种方法是提供字面值，例如：

```
switch (myInteger)
{
    case 1:
        <code to execute if myInteger == 1 >
        break;
    case -1:
        <code to execute if myInteger == -1 >
        break;
    default:
        <code to execute if myInteger != comparisons>
        break;
}
```

另一种方式是使用常量。常量与其他变量一样，但有一个重要的区别：它们包含的值是固定不变的。一旦给常量指定一个值，该常量在代码执行的过程中，其值一直不变。在这里使用常量是很方便的，因为它们通常更便于阅读，在比较时，看不到要比较的实际值。

声明常量需要指定变量类型和关键字 `const`，同时必须给它们赋值，例如：



```
const int intTwo = 2;
```

这行代码是有效的，但如果编写如下代码：

```
const int intTwo;
intTwo = 2;
```

就会产生一个编译错误。如果在最初的赋值之后，试图通过任何方式改变常量的值，也会出现编译错误。

在下面的示例中，将使用 switch 语句，根据用户为测试字符串输入的值，将不同的字符串写到控制台上。

### 试一试：使用 switch 语句

- (1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新控制台应用程序 Ch04Ex03。
- (2) 把下述代码添加到 Program.cs 中：



```
static void Main(string[] args)
{
    const string myName = "karli";
    const string sexyName = "angelina";
    const string sillyName = "ploppy";
    string name;
    Console.WriteLine("What is your name?");
    name = Console.ReadLine();
    switch (name.ToLower ())
    {
        case myName:
            Console.WriteLine("You have the same name as me!");
            break;
        case sexyName:
            Console.WriteLine("My, what a sexy name you have!");
            break;
        case sillyName:
            Console.WriteLine("That's a very silly name.");
            break;
    }
    Console.WriteLine("Hello {0}!", name);
    Console.ReadKey();
}
```

代码段 Ch04Ex03\Program.cs

- (3) 执行代码，输入一个姓名，结果如图 4-3 所示。

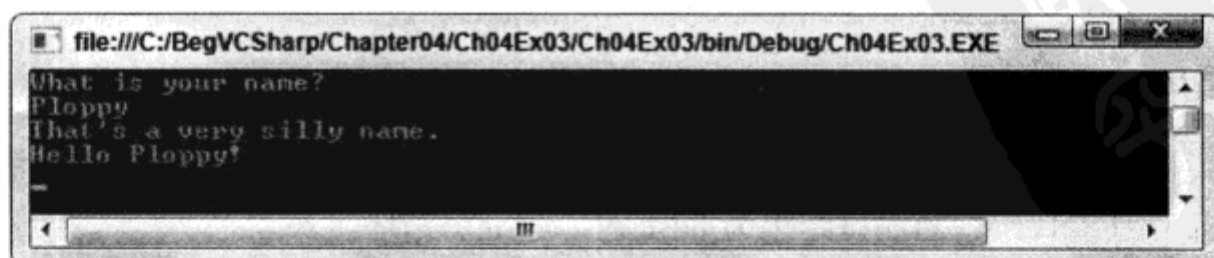


图 4-3

### 示例的说明

这段代码建立了 3 个常量字符串，接受用户输入的一个字符串，再根据输入的字符串把文本写到控制台上。这里，字符串是用户输入的姓名。

在比较输入的姓名(在变量 `name` 中)和常量值时，首先要用 `name.ToLower()` 把输入的姓名转换为小写。`name.ToLower()` 是一个标准命令，可用于处理所有的字符串变量，在不能确定用户输入的内容时，使用它是很方便的。使用这个技术，字符串 `Karli`、`kArLi`、`karli` 等就会与测试字符串 `karli` 匹配了。

`switch` 语句尝试将输入的字符串与定义的常量值进行匹配，如果成功，就会用一条个性化的消息问候用户。如果不匹配，则只简单地问候用户。

`switch` 语句对 `case` 语句的数量上没有限制，所以可以扩展这段代码，使之包含自己能想到的每个姓名，但这需要耗费一些时间。

## 4.4 循环

循环就是重复执行语句。这个技术使用起来非常方便，因为可以对操作重复任意多次(上千次，甚至百万次)，而无需每次都编写相同的代码。

例如，下面的代码计算一个银行账户在 10 年后的金额，假定支付每年的利息，且该账户没有其他款项的存取：

```
double balance = 1000;
double interestRate = 1.05; // 5% interest/year
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

将相同代码编写 10 次很费时间，如果把 10 年改为其他值，又会如何？那就必须把该代码行手工复制需要的次数，这是一件多么痛苦的事！幸运的是，完全不必这样做。使用一个循环就可以对指令执行需要的次数。

循环的另一个重要类型是一直循环到给定的条件满足为止。这些循环比上面描述的循环稍简单些(但也是很有效的)，所以首先从这类循环开始。

### 4.4.1 do 循环

`do` 循环以下述方式执行：执行标记为循环的代码，然后进行一个布尔测试，如果测试的结果为 `true`，就再次执行这段代码。当测试结果为 `false` 时，就退出循环。

`do` 循环的结构如下：

```
do
{
    <code to be looped>
} while (<Test>);
```

其中计算<Test>会得到一个布尔值。



while 语句之后必须使用分号。

例如，使用该结构可以把从 1~10 的数字输出到一列上：

```
int i = 1;
do
{
    Console.WriteLine("{0}", i++);
} while (i <= 10);
```

在把 i 的值写到屏幕上后，使用后缀形式的++运算符递增 i 的值，所以需要检查一下 i <= 10，把 10 也包含在输出到控制台的数字中。

下面的示例使用这个结构略微修改一下本节引言中的代码。该段代码计算了一个账户在 10 年后的余额。这次使用一个循环，根据起始的金额和固定利率，计算该账户的金额要花多长时间才能达到某个指定的数值。

#### 试一试：使用 do 循环

- (1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新的控制台应用程序 Ch04Ex04。
- (2) 把下述代码添加到 Program.cs 中：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    double balance, interestRate, targetBalance;
    Console.WriteLine("What is your current balance?");
    balance = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("What is your current annual interest rate (in %)?");
    interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
    Console.WriteLine("What balance would you like to have?");
    targetBalance = Convert.ToDouble(Console.ReadLine());
    int totalYears = 0;
    do
    {
        balance *= interestRate;
        ++totalYears;
    }
    while (balance < targetBalance);
    Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
        totalYears, totalYears == 1 ? "" : "s", balance);
    Console.ReadKey();
}
```

代码段 Ch04Ex04\Program.cs

(3) 执行代码，输入一些值，示例结果如图 4-4 所示。

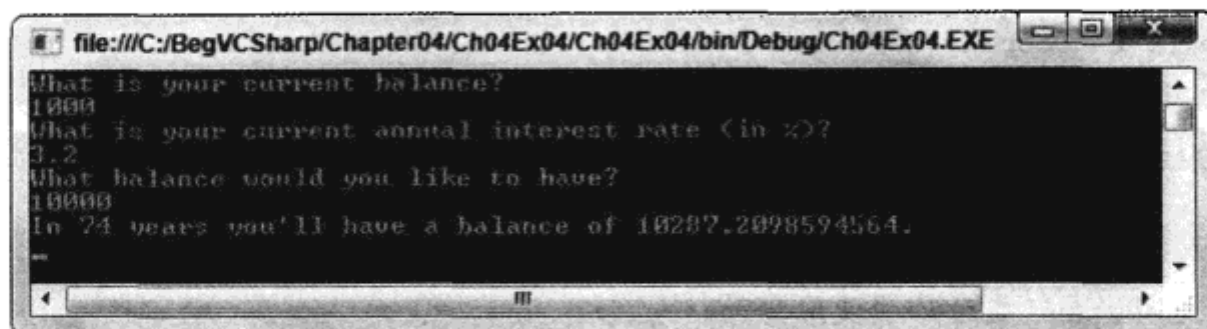


图 4-4

#### 示例的说明

这段代码利用固定的利率，对年度计算余额的过程重复必要的次数，直到满足临界条件为止。在每次循环中，递增一个计数器变量，就可以确定需要多少年：

```
int totalYears = 0;
do
{
    balance *= interestRate;
    ++totalYears;
}
while (balance < targetBalance);
```

然后就可以将这个计数器变量用作输出结果的一部分：

```
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
    totalYears, totalYears == 1 ? "" : "s", balance);
```



这可能是?: (三元)运算符最常见的用法了——用最少的代码有条件地格式化文本。如果 totalYears 不等于 1，就在 year 后面输出一个 s。

但这段代码并不完美，考虑一下目标余额少于当前余额的情况，则结果应如图 4-5 所示。

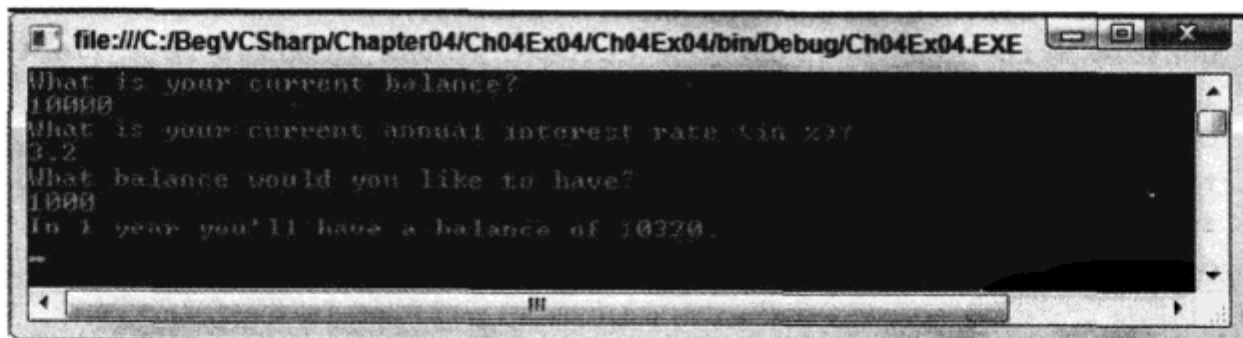


图 4-5

do 循环至少要执行一次。有时(像这种情况)这并不是很理想。当然，可以添加一个 if 语句。

```
int totalYears = 0;
if (balance < targetBalance)
{
    do
    {
```

```

        balance *= interestRate;
        ++totalYears;
    }
    while (balance < targetBalance);
}
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
    totalYears, totalYears == 1? "" : "s", balance);

```

这显然增加了不必要的复杂性。更好的解决方案是使用 `while` 循环。

#### 4.4.2 while 循环

`while` 循环非常类似于 `do` 循环，但有一个明显的区别：`while` 循环中的布尔测试是在循环开始时进行，而不是最后。如果测试结果为 `false`，就不会执行循环。程序会直接跳转到循环之后的代码。

按下述方式指定 `while` 循环：

```

while (<Test>)
{
    <code to be looped>
}

```

它使用的方式与 `do` 循环几乎完全相同，例如：

```

int i = 1;
while (i <= 10)
{
    Console.WriteLine("{0}", i++);
}

```

这段代码的执行结果与前面的 `do` 循环相同，它在一列中输出从 1~10 的数字。下面使用 `while` 循环修改上一个示例。

#### 试一试：使用 while 循环

- (1) 在目录 `C:\BegVCSharp\Chapter04` 中创建一个新的控制台应用程序 `Ch04Ex05`。
- (2) 修改代码，如下所示(开头使用 `Ch04Ex04` 中的代码，记住删除原来 `do` 循环最后的 `while` 语句)：



可从  
wrox.com  
下载源代码

```

static void Main(string[] args)
{
    double balance, interestRate, targetBalance;
    Console.WriteLine("What is your current balance?");
    balance = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("What is your current annual interest rate (in %)?");
    interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
    Console.WriteLine("What balance would you like to have?");
    targetBalance = Convert.ToDouble(Console.ReadLine());
    int totalYears = 0;
    while (balance < targetBalance)
    {
        balance *= interestRate;
        ++totalYears;
    }
}

```

```
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
    totalYears, totalYears == 1 ? "" : "s", balance);
    if (totalYears == 0)
        Console.WriteLine(
            "To be honest, you really didn't need to use this calculator.");
    Console.ReadKey();
}
```

代码段 Ch04Ex05\Program.cs

(3) 再次执行代码，但这次使用少于起始余额的目标余额，如图 4-6 所示。

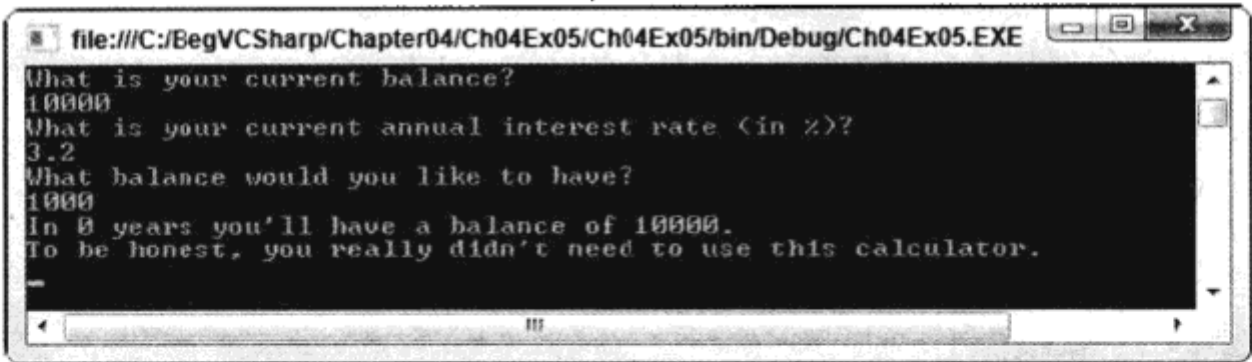


图 4-6

示例的说明

这段代码只是把 do 循环改为 while 循环，就解决了上一个示例中的问题。把布尔测试移到开头，就考虑了不需要执行循环的情况，可以直接跳转到输出结果上。

当然，这种情况还有一个解决方案。例如，可以检查用户输入，确保目标余额大于起始余额。此时，可以把用户输入部分放在循环中，如下所示：

```
Console.WriteLine("What balance would you like to have?");
do
{
    targetBalance = Convert.ToDouble(Console.ReadLine());
    if (targetBalance <= balance)
        Console.WriteLine("You must enter an amount greater than " +
            "your current balance!\nPlease enter another value.");
}
while (targetBalance <= balance);
```

这将拒绝接受无意义的值，得到如图 4-7 所示的结果。

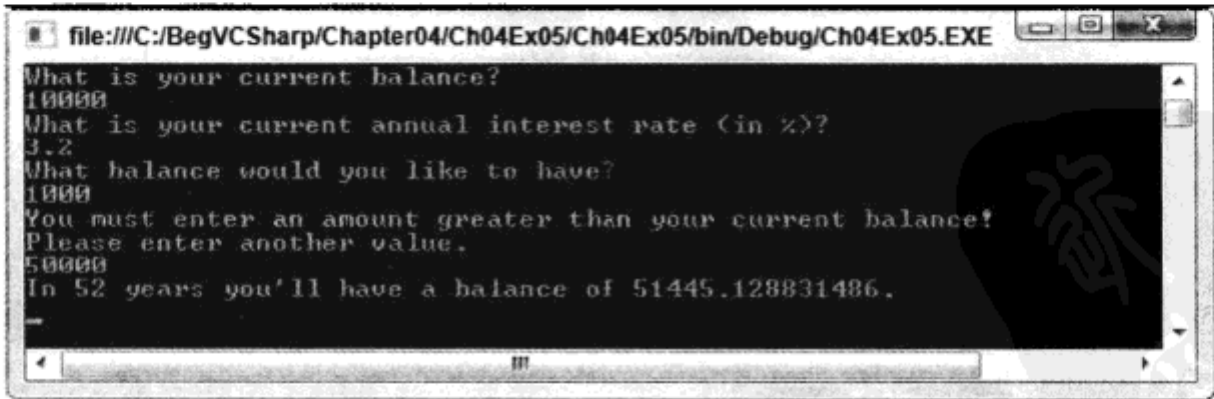


图 4-7

在设计应用程序时，用户输入的有效性检查是一个很重要的主题，本书将提供更多这方面的示例。



### 4.4.3 for 循环

本章介绍的最后一类循环是 for 循环。这类循环可以执行指定的次数，并维护它自己的计数器。要定义 for 循环，需要下列信息：

- 初始化计数器变量的一个起始值。
- 继续循环的条件，它应涉及到计数器变量。
- 在每次循环的最后，对计数器变量执行一个操作。

例如，如果要在循环中，使计数器从 1 递增到 10，递增量为 1，则起始值为 1，条件是计数器小于或等于 10，在每次循环的最后，要执行的操作是给计数器加 1。

这些信息必须放在 for 循环的结构中，如下所示：

```
for (<initialization>; <condition>; <operation>)
{
    <code to loop>
}
```

它的工作方式与下述 while 循环完全相同：

```
<initialization>
while (<condition>)
{
    <code to loop>
    <operation>
}
```

但 for 循环的格式使代码更易于阅读，因为其语法是在一个地方包括循环的全部规则，而不是把几个语句放在代码的不同地方。

前面使用 do 和 while 循环输出了从 1~10 的数字。下面看看如何使用 for 循环完成这个任务：

```
int i;
for (i = 1; i <= 10; ++i)
{
    Console.WriteLine("{0}", i);
}
```

计数器变量是一个整数 i，它的初始值是 1，在每次循环的最后递增 1。在每次循环过程中，把 i 的值写到控制台上。

注意，当 i 的值为 11 时，将执行循环后面的代码。这是因为在 i 等于 10 的循环末尾，i 会递增为 11。这是在测试条件  $i \leq 10$  之前发生的，此时循环结束。与 while 循环一样，在第一次执行前，只在条件测定为 true 时才执行 for 循环，所以可能根本就不会执行循环中的代码。

最后要注意的是，可以把计数器变量声明为 for 语句的一部分，重新编写上述代码，如下所示：

```
for (int i = 1; i <= 10; ++i)
{
    Console.WriteLine("{0}", i);
}
```

但如果这么做，就不能在循环外部使用变量 i (参见第 6 章中的“变量作用域”一节)。

下面介绍一个使用 for 循环的示例。我们已经使用了几个循环，所以这个示例比较有趣：它将

显示一个 Mandelbrot 集合(使用纯文本字符, 看起来不会那么吸引人)!

### 试一试: 使用 for 循环

- (1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新的控制台应用程序 Ch04Ex06。
- (2) 将以下代码添加到 Program.cs 中:



```
static void Main(string[] args)
{
    double realCoord, imagCoord;
    double realTemp, imagTemp, realTemp2, arg;
    int iterations;
    for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
    {
        for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
        {
            iterations = 0;
            realTemp = realCoord;
            imagTemp = imagCoord;
            arg = (realCoord * realCoord) + (imagCoord * imagCoord);
            while ((arg < 4) && (iterations < 40))
            {
                realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
                    - realCoord;
                imagTemp = (2 * realTemp * imagTemp) - imagCoord;
                realTemp = realTemp2;
                arg = (realTemp * realTemp) + (imagTemp * imagTemp);
                iterations += 1;
            }
            switch (iterations % 4)
            {
                case 0:
                    Console.Write(".");
                    break;
                case 1:
                    Console.Write("o");
                    break;
                case 2:
                    Console.Write("O");
                    break;
                case 3:
                    Console.Write("@");
                    break;
            }
        }
        Console.Write("\n");
    }
    Console.ReadKey();
}
```

代码段 Ch04Ex06\Program.cs

- (3) 执行代码, 结果如图 4-8 所示。

示例的说明

这里不打算详细说明如何计算 Mandelbrot 集合，而是解释为什么需要在这段代码中使用循环。如果你对数学不感兴趣，可以快速浏览下面两段，因为它们对代码的理解非常重要。

Mandelbrot 集合中的每个位置都对应于公式  $N = x + y*i$  中的一个复数。实数部分是  $x$ ，虚数部分是  $y$ ， $i$  是  $-1$  的平方根。图像中各个位置的  $x$  和  $y$  坐标对应于复数的  $x$  和  $y$  部分。

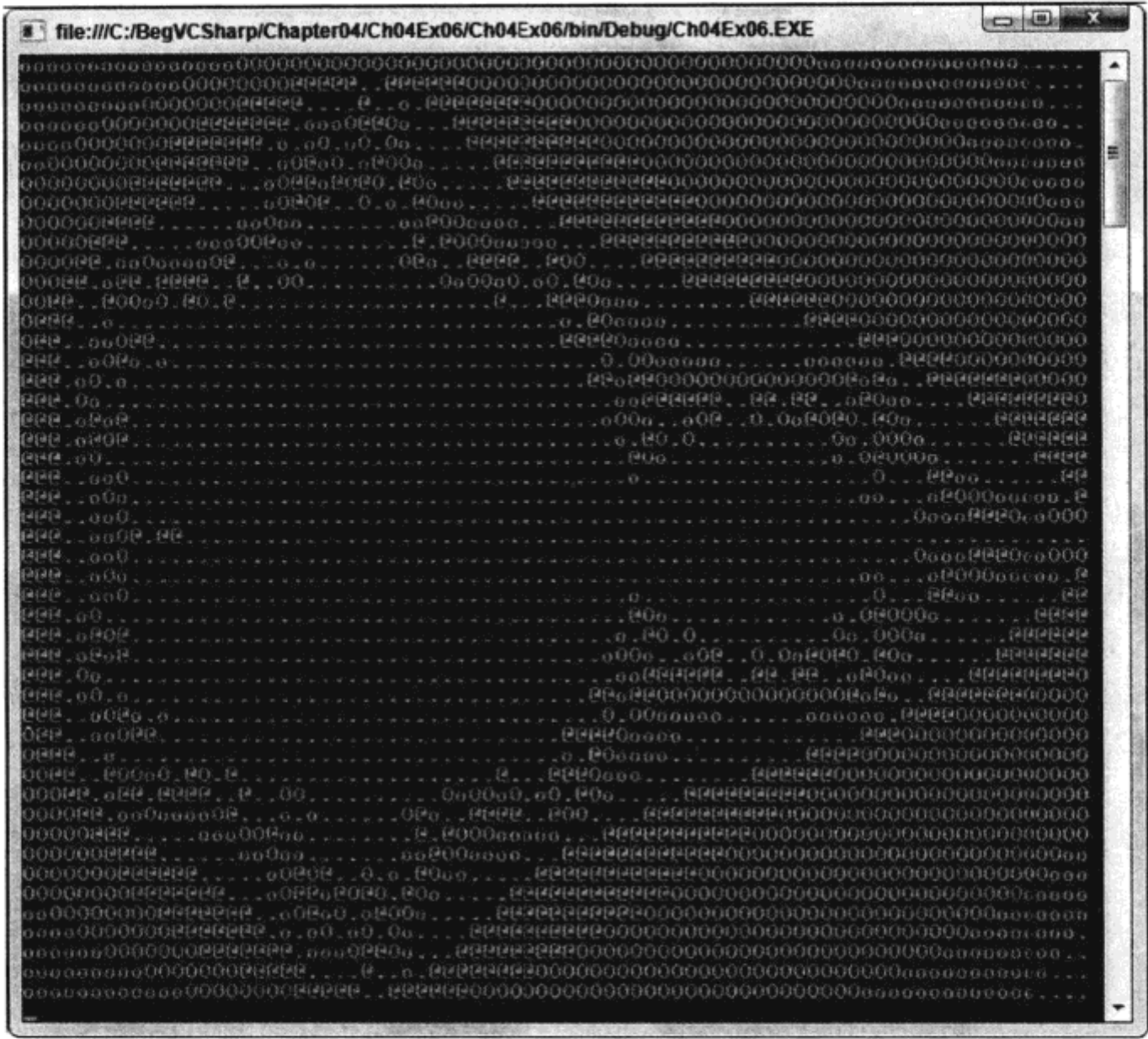


图 4-8

图像中的每个位置用参数  $N$  来表示，它是  $x*x + y*y$  的平方根。如果这个值大于或等于 2，则这个数字对应的位置值是 0。如果参数  $N$  的值小于 2，就把  $N$  的值改为  $N*N - N$  (即  $N = (x*x - y*y - x) + (2*x*y - y)*i$ )，并再次测试这个新  $N$  值。如果这个值大于或等于 2，则这个数字对应的位置值是 1。这个过程将一直继续下去，直到给图像中的位置赋一个值，或迭代执行的次数超过指定的次数为止。

根据给图像中每个点赋予的值，在图形环境下，屏幕上会显示某种颜色的像素。但是，本例使用的是文本环境，所以屏幕上显示的是一个字符。

下面看看代码，以及其中的循环。首先声明计算过程中需要的变量：

```
double realCoord, imagCoord;
double realTemp, imagTemp, realTemp2, arg;
int iterations;
```

其中 `realCoord` 和 `imagCoord` 是 `N` 的实数和虚数部分, 其他 `double` 变量是计算过程中的临时信息。`Iterations` 记录在参数 `N(arg)` 等于或大于 2 之前的迭代次数。

接着是两个 `for` 循环, 迭代图像中所有点的坐标(使用比++ 或--略复杂一些的语法来修改计数器, 这是一种常见的功能强大的技术):

```
for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
{
    for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
    {
```

这里选择合适的边界来显示 Mandelbrot 图像的主要部分。如果要放大这个图像, 可以放大这些边界。

在这两个循环中, 代码处理 Mandelbrot 图像中的一个点, 给 `N` 指定一个值, 这段代码执行要求的迭代计算, 给定当前点的测试值。

首先初始化一些变量:

```
iterations = 0;
realTemp = realCoord;
imagTemp = imagCoord;
arg = (realCoord * realCoord) + (imagCoord * imagCoord);
```

接着用 `while` 循环执行迭代。使用 `while` 循环, 而不是 `do` 循环, 是为了防止 `N` 的初始值大于 2, 如果 `N` 大于 2, `iterations = 0` 就是需要的答案, 不再需要计算了。

注意这里没有全面计算参数, 而仅获取  $x^2 + y^2$  的值, 并检查该值是否小于 4。这样简化了计算, 因为 2 是 4 的平方根, 不需要计算平方根。

```
while ((arg < 4) && (iterations < 40))
{
    realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
               - realCoord;
    imagTemp = (2 * realTemp * imagTemp) - imagCoord;
    realTemp = realTemp2;
    arg = (realTemp * realTemp) + (imagTemp * imagTemp);
    iterations += 1;
}
```

这个循环计算上述的数值, 其最大迭代数是 40。

把当前点的值存储在 `iterations` 中后, 再使用 `switch` 语句选择要输出的字符。这里只使用 4 个不同字符, 而不是 40 个, 且使用求余运算符(%), 这样 0、4、8 等使用一个字符, 1、5、9 等使用另一个字符, 依次类推:

```
switch (iterations % 4)
{
    case 0:
        Console.Write(".");
        break;
    case 1:
        Console.Write("o");
        break;
    case 2:
```

```

        Console.Write("0");
        break;
    case 3:
        Console.Write("@");
        break;
}

```

注意这里使用的是 `Console.Write()`，而不是 `Console.WriteLine()`，因为每次输出一个字符时，并不需要从一个新行开始。在最内层的一个 `for` 循环结束后，需要结束一行，所以使用前面介绍的转义序列输出行结束符。

```

    }
    Console.Write("\n");
}

```

这样，每行都与下一行分隔开来，并进行适当的排列。这个应用程序的最终结果尽管不是很漂亮，也能给人留下深刻的印象。它说明了循环和分支的用途。

#### 4.4.4 循环的中断

有时需要更精细地控制循环代码的处理。C#为此提供了4个命令，其中的3个已经在其他情形中介绍过了：

- **break**——立即终止循环。
- **continue**——立即终止当前的循环(继续执行下一次循环)。
- **goto**——可以跳出循环，到已标记好的位置上(如果希望代码易于阅读和理解，最好不要使用该命令)。
- **return**——跳出循环及其包含的函数(参见第6章)。

**break** 命令可退出循环，继续执行循环后面的第一行代码，例如：

```

int i = 1;
while (i <= 10)
{
    if (i == 6)
        break;
    Console.WriteLine("{0}", i++);
}

```

这段代码输出 1~5 的数字，因为 **break** 命令在 `i` 的值为 6 时退出循环。

**continue** 仅终止当前的循环，而不是整个循环，例如：

```

int i;
for (i = 1; i <= 10; i++)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine(i);
}

```

在上面的示例中，只要 `i` 除以 2 的余数是 0，**continue** 语句就终止当前的循环，所以只显示数字 1、3、5、7 和 9。

第 3 个方法使用前面的 `goto` 语句，例如：

```
int i = 1;
while (i <= 10)
{
    if (i == 6)
        goto exitPoint;
    Console.WriteLine("{0}", i++);
}
Console.WriteLine("This code will never be reached.");
exitPoint:
Console.WriteLine("This code is run when the loop is exited using goto.");
```

注意，使用 `goto` 语句退出循环是合法的(但会有点杂乱)，但使用 `goto` 语句从外部进入循环是非法的。

#### 4.4.5 无限循环

可以通过编写错误代码或错误的设计，定义永不终止的循环，即所谓的无限循环。例如，下面的代码：

```
while (true)
{
    // code in loop
}
```

有时这种代码也是有用的，使用 `break` 语句或者手工使用 Windows 任务管理器总是可以退出这样的循环。但是，当这种情形偶然出现时，就会出问题。考虑下面的循环，它与上一节的 `for` 循环非常类似：

```
int i = 1;
while (i <= 10)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine("{0}", i++);
}
```

`i` 是在循环的最后一行代码执行完后才递增的，即在 `continue` 语句执行完后递增。但在执行到这个 `continue` 语句(此时 `i` 为 2)时，程序会用相同的 `i` 值进行下一个循环，然后测试这个 `i` 值，继续循环，一直这样下去。这就冻结了应用程序。注意仍可以用一般方式退出已冻结的应用程序，所以此时不必重新启动计算机。

### 4.5 小结

本章介绍了可以在代码中使用的各种结构，扩展了您的编程知识。在开始编写更复杂的应用程序时，这些结构的正确使用是非常重要的。

首先用一定的篇幅介绍了布尔逻辑，以及一些按位逻辑的知识。在学习了本章的其他内容后，



再回过头来看看这些逻辑，可以确信，在谈到执行程序中的分支和循环代码时，这个主题是非常重要的。熟悉本节讨论的运算符和技术是很有必要的。

分支结构可以有条件地执行代码，当分支与循环一起使用时，可以在 C#代码中创建出比较复杂的结构。把循环嵌套起来，再放在 if 结构中，就会发现代码的缩进是非常有用的。如果把所有代码都移到屏幕左端，就很难分析它们了，甚至难以调试。此时应确保代码的缩进——用户在以后使用时即可体会到它的种种优势。VS 为此做了大量的工作，但最好在输入代码时进行缩进。

第 5 章将深入探讨变量。

4.6 练习

- (1) 如果两个整数存储在变量 var1 和 var2 中，该进行什么样的布尔测试，看看其中的一个(但不是两个)是否大于 10?
- (2) 编写一个应用程序，其中包含练习(1)中的逻辑，要求用户输入两个数字，并显示它们，但拒绝接受两个数字都大于 10 的情况，并要求用户重新输入。
- (3) 下面的代码存在什么错误?

```
int i;
for (i = 1; i <= 10; i++)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine(i);
}
```

- (4) 修改 Mandelbrot 集合应用程序，要求用户输入图像的边界，显示选中的图像部分。当前代码输出的字符应正好能放在控制台应用程序的一行上。考虑如何使每个选中的图像正好占据大小相同的空间，以最大化可视区域。

附录 A 给出了练习答案。

4.7 本章要点

主 题	重 要 概 念
布尔逻辑	布尔逻辑使用布尔值(true 和 false)计算条件。布尔运算符用于比较数值，返回布尔结果。一些布尔运算符也用于对数值的底层位结构执行按位操作，还有一些专门的按位运算符
分支	可以使用布尔逻辑控制程序流。计算为布尔值的表达式可以用于确定是否执行某个代码块，可以使用 if 语句或?: (三元)运算符进行简单的分支，或者使用 switch 语句同时检查多个条件
循环	循环允许根据指定的条件多次执行代码块。使用 do 和 while 循环可以在布尔表达式为 true 时执行代码，使用 for 循环可以在循环代码中包含一个计数器。循环可以使用 continue 中断当前的迭代，或者使用 break 完全中断。一些循环只能在用户强制中断时结束，它们称为无限循环



# 变量的更多内容

## 本章内容:

---

- 如何在类型之间进行隐式和显式转换
- 如何创建和使用枚举类型
- 如何创建和使用结构类型
- 如何创建和使用数组
- 如何处理字符串值

前面介绍了有关 C# 语言的一些内容，现在将回顾和讨论与变量相关的其他一些较复杂的论题。

首先要讨论的主题是类型转换，即把值从一种类型转换为另一种类型。前面已经描述了其中的一些信息，这里则要正式讨论。掌握这个论题可以更好地理解表达式中(有意或无意)混合使用的类型，更好地控制处理数据的方式。这有助于理顺代码，避免引起不必要的误解。

接着阐述另外一些类型的变量：

- **枚举**——变量类型，用户定义了一组可能的离散值，这些值可以用人们能理解的方式使用。
- **结构**——合成的变量类型，由用户定义的一组其他变量类型组成。
- **数组**——包含一种类型的多个变量，可以以索引方式访问各个数值。

这些类型比前面使用的简单类型复杂一些，但可以使工作更容易完成。最后，学习另一个与字符串相关的主题——基本字符串处理。

## 5.1 类型转换

本书前面说过，无论是什么类型，所有的数据都是一系列的位，即一系列 0 和 1。变量的含义是通过解释这些数据的方式来传达的。最简单的示例是 `char` 类型，这种类型用一个数字表示 Unicode 字符集中的一个字符。实际上，这个数字与 `ushort` 的存储方式完全相同——它们都存储 0~65535 之间的数字。

但一般情况下，不同类型的变量使用不同的模式来表示数据。这意味着，即使可以把一系列的

位从一种类型的变量移动到另一种类型的变量中(也许它们占用的存储空间相同,也许目标类型有足够的存储空间包含所有的源数据位),结果也可能与期望的不同。

这并不是数据位从一个变量到另一个变量的一对一映射,而是需要对数据进行类型转换。类型转换采用以下两种形式:

- **隐式转换:** 从类型 A 到类型 B 的转换可以在所有情况下进行,执行转换的规则非常简单,可以让编译器执行转换。
- **显式转换:** 从类型 A 到类型 B 的转换只能在某些情况下进行,转换的规则比较复杂,应进行某种类型的处理。

5.1.1 隐式转换

隐式转换不需要做任何工作,也不需要另外编写代码。考虑下面的代码:

```
var1 = var2;
```

如果var2的类型可以隐式地转换为var1的类型,这个赋值语句就涉及到一个隐式转换。它也可能只处理相同类型的两个变量,不需要隐式转换。例如,ushort和char的值是可以互换的,因为它们都可以存储0~65535之间的数字,在这两个类型之间可以进行隐式转换,如下面的代码所示:

```
ushort destinationVar;  
char sourceVar = 'a';  
destinationVar = sourceVar;  
Console.WriteLine("sourceVar val: {0}", sourceVar);  
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

这里存储在sourceVar中的值放在destinationVar中。在用两个Console.WriteLine()命令输出变量时,得到如下结果:

```
sourceVar val: a  
destinationVar val: 97
```

即使两个变量存储的是相同的信息,使用不同的类型解释它们时,方式也是不同的。

简单类型有许多隐式转换;bool和string没有隐式转换,但数值类型有一些隐式转换。表 5-1 列出了编译器可以隐式执行的数值转换(记住,char存储的是数值,所以char被当作一个数值类型)。

表 5-1

类 型	可以安全地转换为
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal

(续表)

类 型	可以安全地转换为
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

不要担心——不需要记住这个表格，因为很容易看出编译器可以执行哪些隐式转换。第 3 章中的一个表列出了每种简单数字类型的取值范围。这些类型的隐式转换规则是：任何类型 A，只要其取值范围完全包含在类型 B 的取值范围内，就可以隐式转换为类型 B。

其原因是很简单的。如果要把一个值放在变量中，而该值超出了变量的取值范围，就会出问题。例如，short 类型的变量可以存储 0~32767 的数字，而 byte 可以存储的最大值是 255，所以如果要把一个 short 值转换为 byte 值，就会出问题。如果 short 包含的值在 256~32767 之间，相应数值就不能放在 byte 中。

但是，如果 short 类型变量中的值小于 255，就应能转换这个值，对吗？答案是可以。具体地说是可以，但必须使用显式转换。执行显式转换有点类似于“我已经知道你对我这么做提出了警告，但我将对其后果负责”。

5.1.2 显式转换

顾名思义，在明确要求编译器把数值从一种数据类型转换为另一种数据类型时，就是在执行显式转换。因此，这需要另外编写代码，代码的格式将随着转换方法而异。在学习显式转换代码前，先分析如果不添加任何显式转换代码，会发生什么情况。

例如，下面对上一节的代码进行修改，试着把 short 值转换为 byte：

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

如果编译这段代码，就会产生如下错误：

```
Cannot implicitly convert type 'short' to 'byte'. An explicit conversion exists
(are you missing a cast?)
```

幸运的是，C#编译器可以检测出没有进行显式转换！

为了成功编译这段代码，需要添加代码，进行显式转换。最简单的方式是把 short 变量强制转换为 byte(由上述错误字符串提出)。强制转换就是强迫数据从一种类型转换为另一种类型，其语法比较简单：

```
<(destinationType)sourceVar>
```

这将把<sourceVar>中的值转换为<destinationType>。



这只在某些情况下是可行的。彼此之间几乎没有什么关系的类型或根本没有关系的类型不能进行强制转换。

因此可以使用这个语法修改示例，把 short 变量强制转换为 byte:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

得到如下结果:

```
sourceVar val: 7
destinationVar val: 7
```

在试图把一个值转换为不合适的变量时，会发生什么呢？修改代码，如下所示:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

结果如下:

```
sourceVar val: 281
destinationVar val: 25
```

会发生什么？看看这两个数字的二进制表示，以及可以存储在 byte 中的最大值 255:

```
281 = 100011001
25 = 000011001
255 = 011111111
```

可以看出，源数据的最左边一位丢失了。这会导致一个问题：数据是何时丢失的？显然，当需要显式地把一种数据类型转换为另一种数据类型时，最好能够了解是否有数据丢失了。如果不知道这些，就会发生严重的问题，例如，记账应用程序或确定火箭飞往月球的轨道的应用程序。

一种方式是简单地检查源变量的值，把它与目标变量的取值范围进行比较。还有另一个技术，迫使系统特别注意运行期间的转换。在将一个值放在一个变量中时，如果该值过大，不能放在该类型的变量中，就会导致溢出，这就需要检查。

这里要用到两个关键字 `checked` 和 `unchecked`，称为表达式的溢出检查上下文。以下述方式使用这两个关键字:

```
checked(expression)
unchecked(expression)
```

下面对上一个示例进行溢出检查:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

在执行这段代码时，程序会崩溃，并显示如图 5-1 所示的错误信息(在 `OverflowCheck` 项目中编译这段代码)。



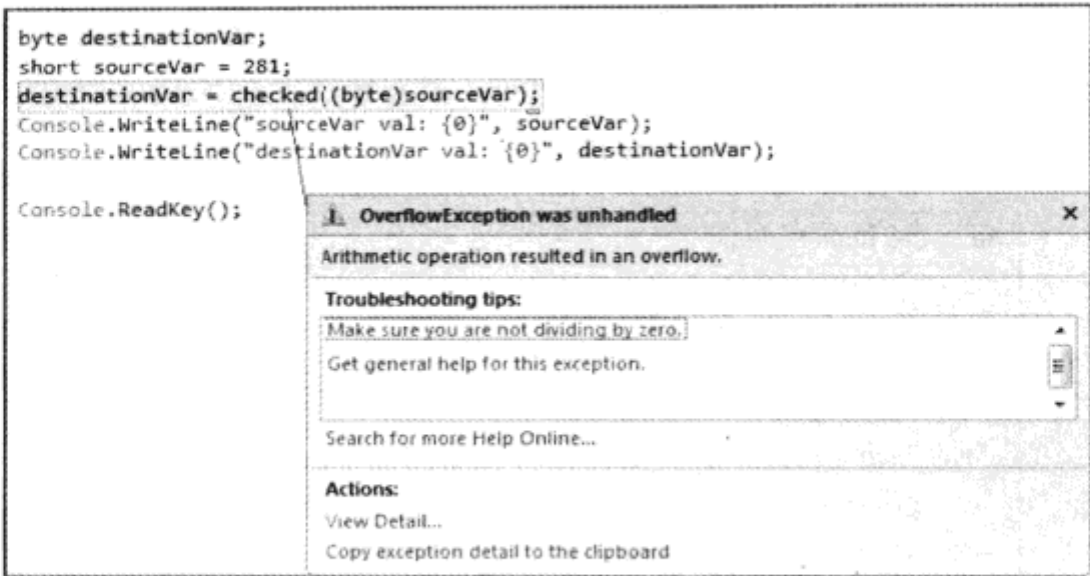


图 5-1

但是，在这段代码中，如果用 `unchecked` 替代 `checked`，就会得到与以前一样的结果，不会出现错误。这与前面的默认做法是一样的。

除了这两个关键字以外，还可以配置应用程序，让这种类型的表达式都包含 `checked` 关键字，除非表达式明确使用 `unchecked` 关键字(换言之，可以改变溢出检查的默认设置)。为此，应修改项目的属性：在 VS 中右击 Solution Explorer 窗口中的项目，选择 `Properties` 选项。单击窗口左边的 `Build`，打开 `Build` 设置，如图 5-2 所示。

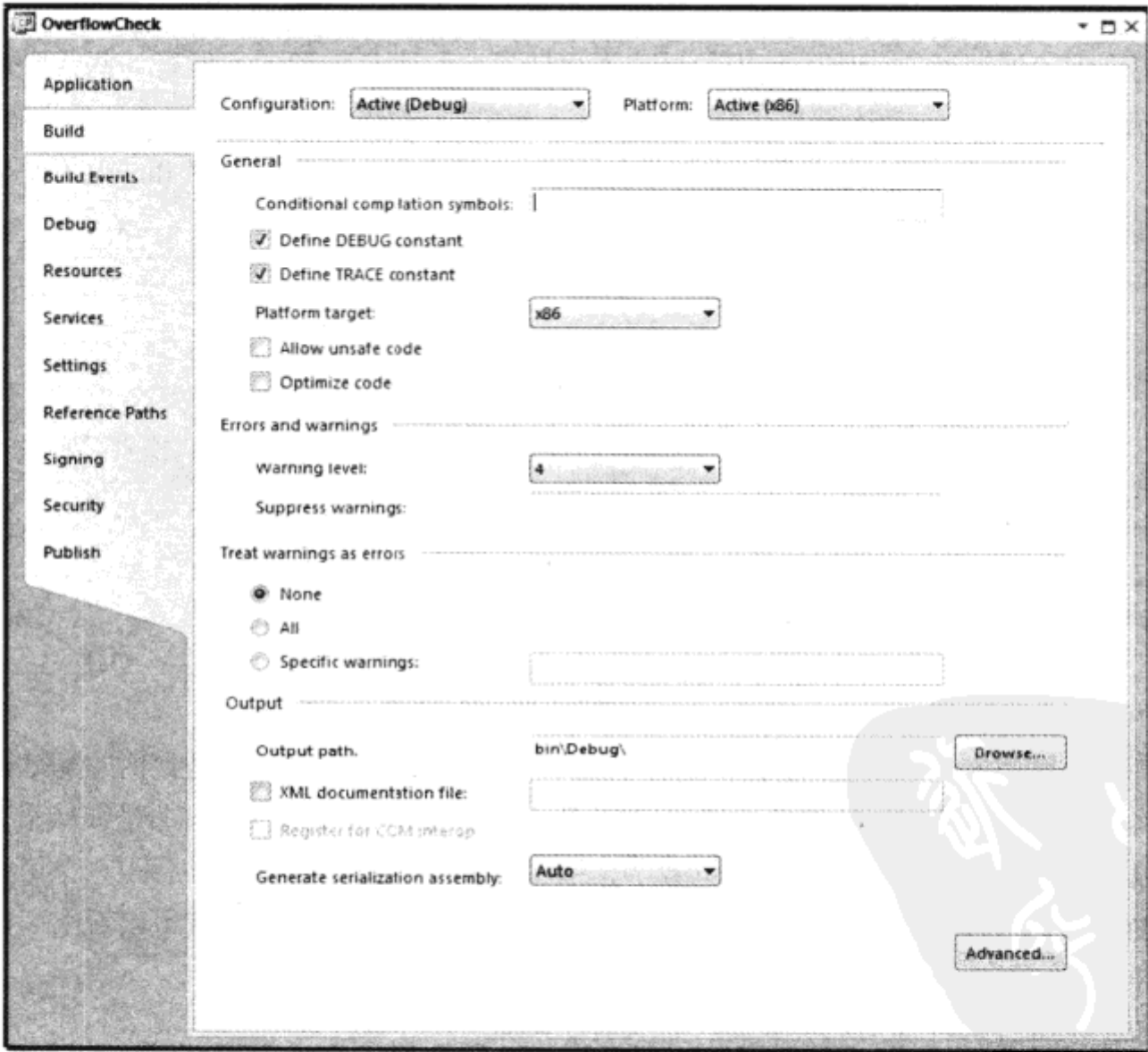


图 5-2

要修改的属性是一个 Advanced 设置，所以单击 Advanced 按钮。在打开的对话框中，选中 Check for arithmetic overflow/underflow 选项，如图 5-3 所示。默认情况下禁用这个设置，激活它可以进行上述 checked 操作。

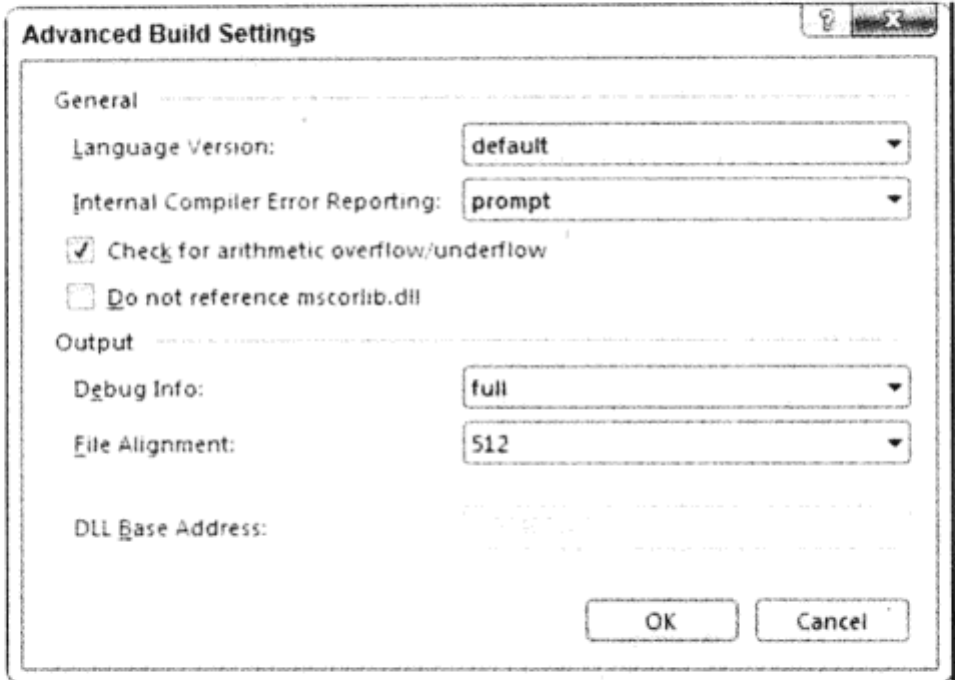


图 5-3

5.1.3 使用 Convert 命令进行显式转换

本书的许多“试一试”示例中使用的显式类型转换，与本章前面的示例有一些区别。前面使用 Convert.ToDouble()等命令把字符串值转换为数值，显然，这种方式并不适用于所有字符串。

例如，如果使用 Convert.ToDouble()把诸如 Number 的字符串转换为一个 double 值，执行代码，就会看到如图 5-4 所示的对话框。

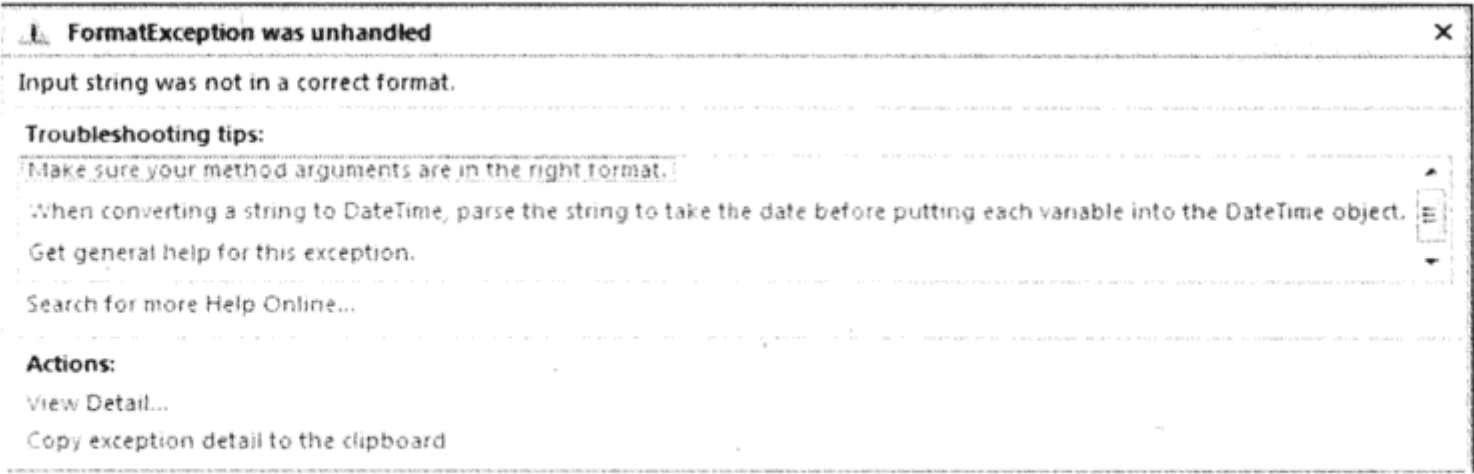


图 5-4

可以看出，执行失败。为了成功执行此类转换，所提供的字符串必须是数值的有效表达方式，该数还必须是不会溢出的数。数值的有效表达方式是：首先是一个可选符号(加号或减号)，然后是 0 位或多位数字，一个句点后跟一位或多位数字，接着是一个可选的 e 或 E，后跟一个可选符号和一位或多位数字(在这个序列之前或之后)和空格。利用这些可选的额外数据，就可以看出 -1.2451e-24 这样复杂的字符串是一个数值。

按这种方式可以进行许多显式转换，如表 5-2 所示。

表 5-2

命 令	结 果
Convert.ToBoolean(val)	val 转换为 bool
Convert.ToByte(val)	val 转换为 byte
Convert.ToChar(val)	val 转换为 char
Convert.ToDecimal(val)	val 转换为 decimal
Convert.ToDouble(val)	val 转换为 double
Convert.ToInt16(val)	val 转换为 short
Convert.ToInt32(val)	val 转换为 int
Convert.ToInt64(val)	val 转换为 long
Convert.ToSByte(val)	val 转换为 sbyte
Convert.ToSingle(val)	val 转换为 float
Convert.ToString(val)	val 转换为 string
Convert.ToUInt16(val)	val 转换为 ushort
Convert.ToUInt32(val)	val 转换为 uint
Convert.ToUInt64(val)	val 转换为 ulong

其中 val 可以是大多数变量类型(如果这些命令不能处理该类型的变量，编译器就会告诉用户)。

但如表 5-2 所示，转换的名称略不同于 C# 类型名称，例如，要转换为 int，应使用 Convert.ToInt32()。这是因为这些命令来自于 .NET Framework 的 System 名称空间，而不是本机 C# 本身。这样它们就可以在除 C# 以外的其他 .NET 兼容语言中使用。

对于这些转换要注意的一个问题是，它们总是要进行溢出检查，checked 和 unchecked 关键字以及项目属性设置不起作用。

下面的示例包括本节介绍的许多转换类型。它声明和初始化许多不同类型的变量，再在它们之间进行隐式和显式转换。

试一试：类型转换的实践

- (1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex01。
- (2) 把下述代码添加到 Program.cs 中：



```
static void Main(string[] args)
{
    short  shortResult, shortVal = 4;
    int    integerVal = 67;
    long   longResult;
    float  floatVal = 10.5F;
    double doubleResult, doubleVal = 99.999;
    string stringResult, stringVal = "17";
    bool   boolVal = true;

    Console.WriteLine("Variable Conversion Examples\n");
}
```

```
doubleResult = floatVal * shortVal;
Console.WriteLine("Implicit, -> double: {0} * {1} -> {2}", floatVal,
    shortVal, doubleResult);

shortResult = (short)floatVal;
Console.WriteLine("Explicit, -> short: {0} -> {1}", floatVal,
    shortResult);

stringResult = Convert.ToString(boolVal) +
    Convert.ToString(doubleVal);
Console.WriteLine("Explicit, -> string: \"{0}\" + \"{1}\" -> {2}",
    boolVal, doubleVal, stringResult);

longResult = integerVal + Convert.ToInt64(stringVal);
Console.WriteLine("Mixed, -> long: {0} + {1} -> {2}",
    integerVal, stringVal, longResult);
Console.ReadKey();
}
```

代码段 Ch05Ex01\Program.cs

(3) 执行代码，结果如图 5-5 所示。

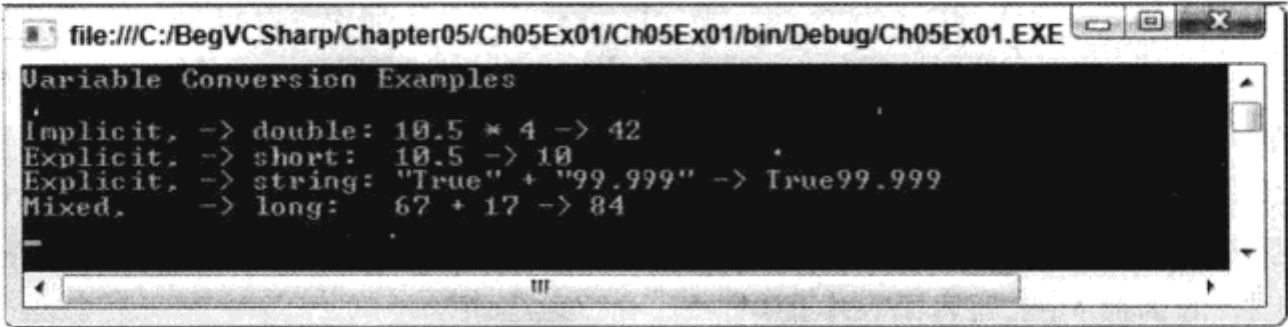


图 5-5

示例的说明

这个示例包含前面介绍的所有转换类型，既有像前面简短代码示例中的简单赋值，也有在表达式中进行的转换。必须考虑这两种情况，因为每个非一元运算符的处理都可能要进行类型转换，而不仅仅是赋值运算符。例如：

```
shortVal * floatVal
```

其中把一个 short 值与一个 float 值相乘。在这样的指令中，没有指定显式转换，应在可能的情况下进行隐式转换。在这个示例中，唯一有意义的隐式转换是把 short 值转换为 float(因为把 float 值转换为 short 需要显式转换)，所以这里使用隐式转换。

也可以重新编写这个过程，使用下述代码：

```
shortVal * (short)floatVal
```



这并不表示两个 short 相乘的结果将返回一个 short 值。因为这个操作很可能大于 32767(这是 short 可以包含的最大值)，所以这个操作的结果实际上是 int。

使用这个数据类型转换语法执行显式转换，其运算符的优先级与其他一元运算符一样，都是优先级中的最高级，如++(用作前缀)。

如果语句涉及混合类型，就根据运算符的优先级，在处理每个运算符时执行转换。这意味着可能出现“中间”转换，例如：

```
doubleResult = floatVal + (shortVal * floatVal);
```

要处理的第一个运算符是\*，如上所述，它将把 shortVal 转换为 float。接着处理+运算符，它不需要进行任何转换，因为这是把两个 float 值相加 (floatVal 和 shortVal \* floatVal 的 float 结果)。在最后处理=运算符时，这个计算的 float 结果转换为 double。

这个转换过程初看起来比较复杂，但只要按照运算符的优先级，把表达式分解为不同的部分，就可以弄明白这个过程。

## 5.2 复杂的变量类型

除了这些简单的变量类型之外，C#还提供了 3 个较复杂(但非常有用)的变量：枚举、结构和数组。

### 5.2.1 枚举

本书迄今介绍的每种类型(除了 string 外)都有明确的取值范围。诚然，有些类型(如 double)的取值范围非常大，可以看作是连续的，却是一个固定的集合。最简单的示例是 bool 类型，它只能取两个值：true 或 false。

有时希望变量提取的是一个固定集合中的值。例如，orientation 类型可以存储 north、south、east 或 west 中的一个值。

此时可以使用枚举类型。枚举就可以完成这个 orientation 类型的任务：它们允许定义一个类型，其中包含提供的限定值集合中的一个值。所以，需要创建自己的枚举类型 orientation，它可以从上述 4 个值中提取一个值。

注意有一个附加的步骤——不是仅仅声明一个给定类型的变量，而是声明和描述一个用户定义的类型，再声明这个新类型的变量。

#### 定义枚举

可以使用 enum 关键字来定义枚举，如下所示：

```
enum <typeName>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

接着声明这个新类型的变量：

```
<typeName> <varName>;
```

并赋值:

```
<varName> = <typeName>.<value>;
```

枚举使用一个基本类型来存储。枚举类型可以提取的每个值都存储为该基本类型的一个值,默认情况下该类型为 int。在枚举声明中添加类型,就可以指定其他基本类型:

```
enum <typeName> : <underlyingType>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

枚举的基本类型可以是 byte、sbyte、short、ushort、int、uint、long 和 ulong。

在默认情况下,每个值都会根据定义的顺序(从 0 开始),自动赋给对应的基本类型值。这意味着 value1 的值是 0, value2 的值是 1, value3 的值是 2 等。可以重写这个赋值过程:使用=运算符,并指定每个枚举的实际值:

```
enum <typeName> : <underlyingType>
{
    <value1> = <actualVal1>,
    <value2> = <actualVal2>,
    <value3> = <actualVal3>,
    ...
    <valueN> = <actualValN>
}
```

还可以使用一个值作为另一个枚举的基础值,为多个枚举指定相同的值:

```
enum <typeName> : <underlyingType>
{
    <value1> = <actualVal1>,
    <value2> = <value1>,
    <value3>,
    ...
    <valueN> = <actualValN>
}
```

没有赋值的任何值都会自动获得一个初始值,这里使用的值是从比上一个明确声明的值大 1 开始的序列。例如,在上面的代码中,<value3>的值是<value1>+1。

注意这可能会产生预料不到的问题,在一个定义(如<value2>=<value1>)后指定的值可能与其他值相同。例如,在下面的代码中,<value4>的值与<value2>相同。

```
enum <typeName> : <underlyingType>
{
    <value1> = <actualVal1>,
    <value2>,
    <value3> = <value1>,
    <value4>,
```



```
...
    <valueN> = <actualValN>
}
```

当然，如果这正是希望的结果，则代码就是正确的。还要注意，以循环方式赋值可能会产生错误，例如：

```
enum <typeName> : <underlyingType>
{
    <value1> = <value2>,
    <value2> = <value1>
}
```

下面看一个示例。其代码定义了一个枚举 `orientation`，然后演示了它的用法。

### 试一试：使用枚举

- (1) 在 `C:\BegVCSharp\Chapter05` 目录中创建一个新控制台应用程序 `Ch05Ex02`。
- (2) 把下列代码添加到 `Program.cs` 中：



```
namespace Ch05Ex02
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east = 3,
        west = 4
    }

    class Program
    {
        static void Main(string[] args)
        {
            orientation myDirection = orientation.north;
            Console.WriteLine("myDirection = {0}", myDirection);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch05Ex02\Program.cs

- (3) 运行应用程序，应得到如图 5-6 所示的输出结果。

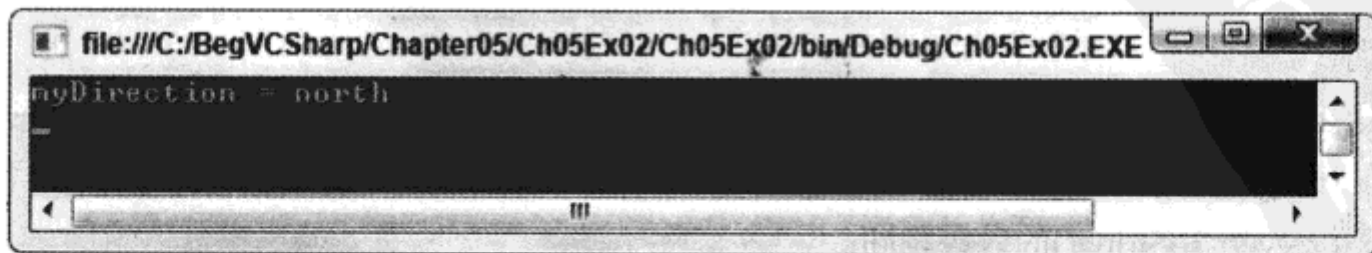


图 5-6

(4) 退出应用程序，修改代码，如下所示：

```
byte directionByte;
string directionString;
orientation myDirection = orientation.north;
Console.WriteLine("myDirection = {0}", myDirection);
directionByte = (byte)myDirection;
directionString = Convert.ToString(myDirection);
Console.WriteLine("byte equivalent = {0}", directionByte);
Console.WriteLine("string equivalent = {0}", directionString);
Console.ReadKey();
```

(5) 再次运行应用程序，输出结果如图 5-7 所示。

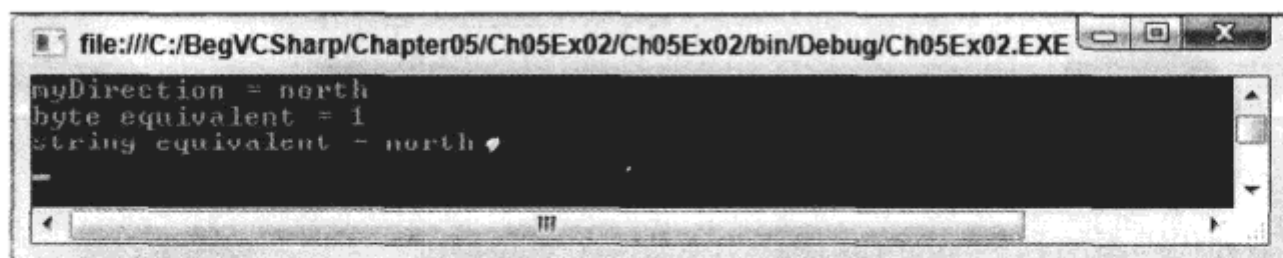


图 5-7

#### 示例的说明

这段代码定义并使用了一个枚举类型 `orientation`。首先要注意的是，类型定义代码放在名称空间 `Ch05Ex02` 中，而没有与其余代码放在一起。这是因为在运行期间，定义代码并不是像执行程序中的代码那样一行一行地执行。应用程序是从已经习惯的位置开始执行的，并可以访问新类型，因为它属于同一个名称空间。

这个示例的第一个迭代演示了创建新类型的变量，给它赋值以及把它输出到屏幕上的基本方法。接着修改代码，把枚举值转换为其他类型。注意这里必须使用显式转换。即使 `orientation` 的基本类型是 `byte`，仍必须使用 `(byte)` 强制类型转换，把 `myDirection` 的值转换为 `byte` 类型：

```
directionByte = (byte)myDirection;
```

如果要将 `byte` 类型转换为 `orientation`，也同样需要进行显式转换。例如，可以使用下述代码将 `byte` 变量 `myByte` 转换为 `orientation`，并把这个值赋给 `myDirection`：

```
myDirection = (orientation)myByte;
```

当然，这里必须小心，因为并不是所有 `byte` 类型变量的值都可以映射为已定义的 `orientation` 值。`orientation` 类型可以存储其他 `byte` 值，所以不会直接产生一个错误，但会在应用程序的后面违反逻辑。

要获得枚举的字符串值，可以使用 `Convert.ToString()`：

```
directionString = Convert.ToString(myDirection);
```

使用 `(string)` 强制类型转换是行不通的，因为需要进行的处理并不仅仅是把存储在枚举变量中的数据放在 `string` 变量中，而是更复杂一些。另外，还可以使用变量本身的 `ToString()` 命令。下面的代码与使用 `Convert.ToString()` 的效果相同：

```
directionString = myDirection.ToString();
```

也可以把 `string` 转换为枚举值，但其语法稍复杂一些。有一个特定的命令用于此类转换，即 `Enum.Parse()`，其用法如下：

```
(enumerationType) Enum.Parse(typeof(enumerationType), enumerationValueString);
```

它使用了另一个运算符 `typeof`，可以得到操作数的类型。对 `orientation` 类型使用这个命令，如下所示：

```
string myString = "north";
orientation myDirection = (orientation) Enum.Parse(typeof(orientation),
                                                    myString);
```

当然，并非所有的字符串值都会映射为一个 `orientation` 值。如果传送的一个值不能映射为枚举值中的一个，就会产生错误。与 C# 中的其他值一样，这些值是区分大小写的，所以如果字符串与一个值相同，但大小写不同(例如，`myString` 设置为 `North`，而不是 `north`)，就会产生错误。

## 5.2.2 结构

下一个要介绍的变量类型是结构(`struct`, `structure` 的简写)。结构就是由几个数据组成的数据结构，这些数据可能具有不同的类型。根据这个结构，可以定义自己的变量类型。例如，假定要存储从起点开始到某一位置的路径，其中路径由一个方向和一个距离值(英里)组成。为简单起见，假定该方向是指南针上的一点(这样，方向就可以用上节的 `orientation` 枚举来表示)，距离值可以用一个 `double` 类型来表示。

通过前面的代码，可以用两个不同的变量来表示该路径：

```
orientation myDirection;
double      myDistance;
```

像这样使用两个变量，是没有错误的，但在一个地方存储这些信息应该更为简单(特别是在需要多个路由时，就更为简单)。

### 定义结构

使用 `struct` 关键字来定义结构，如下所示：

```
struct <typeName>
{
    <memberDeclarations>
}
```

`<memberDeclarations>` 部分包含变量的声明(称为结构的数据成员)，其格式与往常一样。每个成员的声明都采用如下形式：

```
<accessibility> <type> <name>;
```

要让调用结构的代码访问该结构的数据成员，可以对 `<accessibility>` 使用关键字 `public`，例如：

```
struct route
{
    public orientation direction;
    public double      distance;
}
```

定义了结构类型后，就可以定义新类型的变量，来使用该结构：

```
route myRoute;
```

还可以通过句点字符访问这个组合变量中的数据成员：

```
myRoute.direction = orientation.north;
myRoute.distance = 2.5;
```

把这个类型放在下面的“试一试”示例中。其中使用上一节的 `orientation` 枚举和上面的 `route` 结构，然后在代码中处理这个结构，以便您了解结构的工作原理。

### 试一试：使用结构

- (1) 在 `C:\BegVCSharp\Chapter05` 目录中创建一个新控制台应用程序 `Ch05Ex03`。
- (2) 把下列代码添加到 `Program.cs` 中：



可从  
wrox.com  
下载源代码

```
namespace Ch05Ex03
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east = 3,
        west = 4
    }
    struct route
    {
        public orientation direction;
        public double distance;
    }
    class Program
    {
        static void Main(string[] args)
        {
            route myRoute;
            int myDirection = -1;
            double myDistance;
            Console.WriteLine("1) North\n2) South\n3) East\n4) West");
            do
            {
                Console.WriteLine("Select a direction:");
                myDirection = Convert.ToInt32(Console.ReadLine());
            }
            while ((myDirection < 1) || (myDirection > 4));
            Console.WriteLine("Input a distance:");
            myDistance = Convert.ToDouble(Console.ReadLine());
            myRoute.direction = (orientation)myDirection;
            myRoute.distance = myDistance;
            Console.WriteLine("myRoute specifies a direction of {0} and a " +
                "distance of {1}", myRoute.direction, myRoute.distance);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch05Ex03\Program.cs

(3) 执行代码, 输入一个 1~4 之间的数字, 以选择一个方向, 输入一个距离值, 结果如图 5-8 所示。

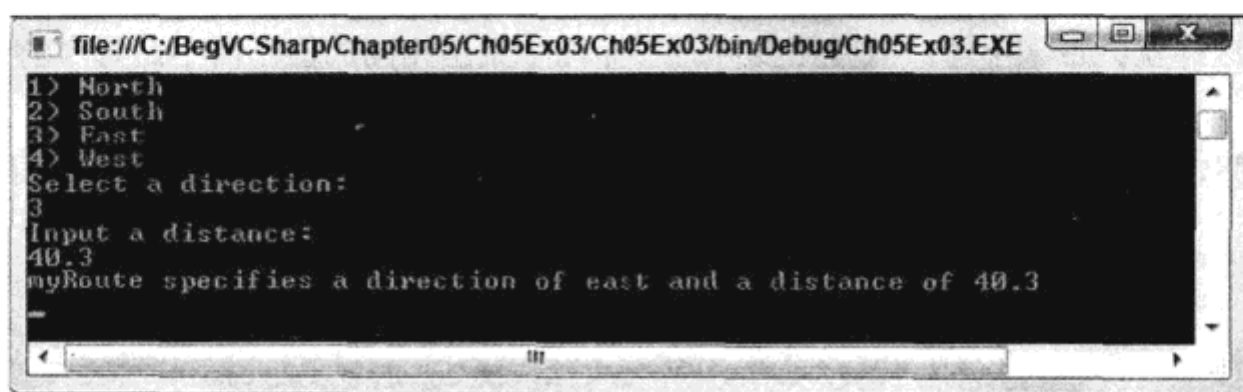


图 5-8

### 示例的说明

结构和枚举一样, 也是在代码的主体之外声明的。在名称空间声明中声明 `route` 结构及其使用的 `orientation` 枚举:

```
enum orientation : byte
{
    north = 1,
    south = 2,
    east = 3,
    west = 4
}
struct route
{
    public orientation direction;
    public double distance;
}
```

代码的主体结构与前面的一些示例类似, 要求用户输入一些信息, 并显示它们。把方向选项放在 `do` 循环中, 对用户的输入进行有效性检查, 拒绝不属于 1~4 范围的整数输入(选择了这样的值后, 它们就会映射到枚举成员上, 以方便赋值)。



不能解释为整数的输入会导致一个错误。本章后面会说明其原因和处理方法。

注意, 在引用 `route` 的成员时, 处理它们的方式与成员类型相同的变量完全一样。赋值语句如下所示:

```
myRoute.direction = (orientation)myDirection;
myRoute.distance = myDistance;
```

可以直接把输入的值放到 `myRoute.distance` 中, 而不会有负面效果, 如下所示:

```
myRoute.distance = Convert.ToDouble(Console.ReadLine());
```

还应进行有效性验证, 但这段代码不存在这一步骤。对结构成员的任何访问都以相同的方式处理。`<structVar>.<memberVar>`形式的表达式可计算`<memberVar>`类型的变量。

### 5.2.3 数组

前面的所有类型都有一个共同点：它们都只存储一个值(结构中存储一组值)。有时，需要存储许多数据，这样就会带来不便。有时需要同时存储几个类型相同的值，而不是每个值使用不同的变量。

例如，假定要对所有朋友的姓名执行一些操作。可以使用简单的字符串变量，如下所示：

```
string friendName1 = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock";
```

但这看起来需要很多工作，特别是需要编写不同的代码来处理每个变量。例如，不能在循环中迭代这个字符串列表。

另一种方式是使用数组。数组是一个变量的索引列表，存储在数组类型的变量中。例如，有一个数组 `friendNames` 存储上述的 3 个名字。在方括号中指定索引，即可访问该数组中的各个成员，如下所示：

```
friendNames[<index>]
```

这个索引是一个整数，第一个条目的索引是 0，第二个条目的索引是 1，依次类推。这样就可以使用循环遍历所有元素，例如：

```
int i;
for (i = 0; i < 3; i++)
{
    Console.WriteLine("Name with index of {0}: {1}", i, friendNames[i]);
}
```

数组有一个基本类型，数组中的各个条目都是这种类型。`friendNames` 数组的基本类型是字符串，因为它要存储 `string` 变量。数组的条目通常称为元素。

#### 1. 声明数组

以下述方式声明数组：

```
<baseType>[] <name>;
```

其中，`<baseType>` 可以是任何变量类型，包括本章前面介绍的枚举和结构类型。数组必须在访问之前初始化，不能像下面这样访问数组或给数组元素赋值：

```
int[] myIntArray;
myIntArray[10] = 5;
```

数组的初始化有两种方式。可以以字面形式指定数组的完整内容，也可以指定数组的大小，再使用关键字 `new` 初始化所有数组元素。

使用字面值指定数组，只需要提供一个用逗号分隔的元素值列表，该列表放在花括号中，例如：

```
int[] myIntArray = {5, 9, 10, 2, 99};
```

其中，`myIntArray` 有 5 个元素，每个元素都被赋予了一个整数值。



另一种方式需要使用下述语法:

```
int[] myIntArray = new int[5];
```

这里使用关键字 **new** 显式地初始化数组, 用一个常量值定义其大小。这种方法会给所有的数组元素赋予同一个默认值, 对于数值类型来说, 其默认值是 0。也可以使用非常量的变量来进行初始化, 例如:

```
int[] myIntArray = new int[arraySize];
```

还可以使用这两种初始化方式的组合:

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

使用这种方式, 数组大小必须与元素个数相匹配。例如, 不能编写如下代码:

```
int[] myIntArray = new int[10] {5, 9, 10, 2, 99};
```

其中数组定义为有 10 个元素, 但只定义了 5 个元素, 所以编译会失败。如果使用变量定义其大小, 该变量必须是一个常量, 例如:

```
const int arraySize = 5;
int[] myIntArray = new int[arraySize] {5, 9, 10, 2, 99};
```

如果省略了关键字 **const**, 运行这段代码就会失败。

与其他变量类型一样, 不见得在声明行中初始化数组。下面的代码是合法的:

```
int[] myIntArray;
myIntArray = new int[5];
```

下面的“试一试”示例利用了本节引言中的示例, 创建并使用一个字符串数组。

### 试一试: 使用数组

- (1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex04。
- (2) 把下列代码添加到 Program.cs 中:



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string[] friendNames = {"Robert Barwell", "Mike Parry",
                           "Jeremy Beacock"};

    int i;
    Console.WriteLine("Here are {0} of my friends:",
                      friendNames.Length);
    for (i = 0; i < friendNames.Length; i++)
    {
        Console.WriteLine(friendNames[i]);
    }
    Console.ReadKey();
}
```

代码段 Ch05Ex04\Program.cs

- (3) 执行代码, 结果如图 5-9 所示。

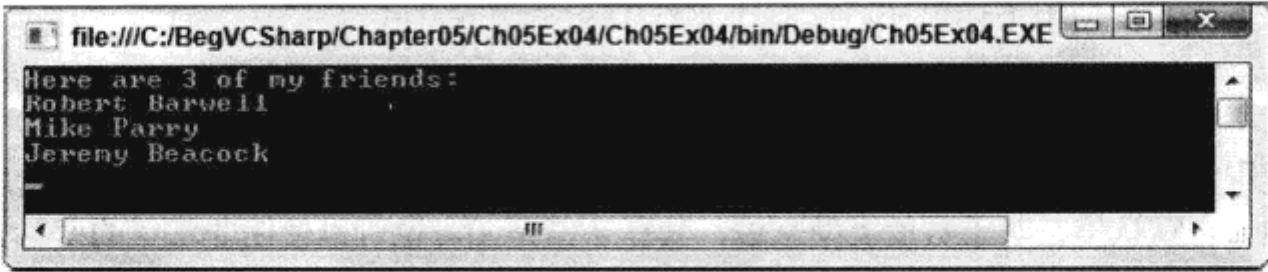


图 5-9

示例的说明

这段代码用 3 个值建立了一个 `string` 数组，并在 `for` 循环中把它们列在控制台上。使用 `friendNames.Length` 来确定数组中元素的个数：

```
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
```

这是获取数组大小的简便方法。在 `for` 循环中输出值容易出错。例如，把 `<` 改为 `<=`，如下所示：

```
for (i = 0; i <= friendNames.Length; i++)
{
    Console.WriteLine(friendNames[i]);
}
```

编译代码，就会弹出如图 5-10 所示的对话框。

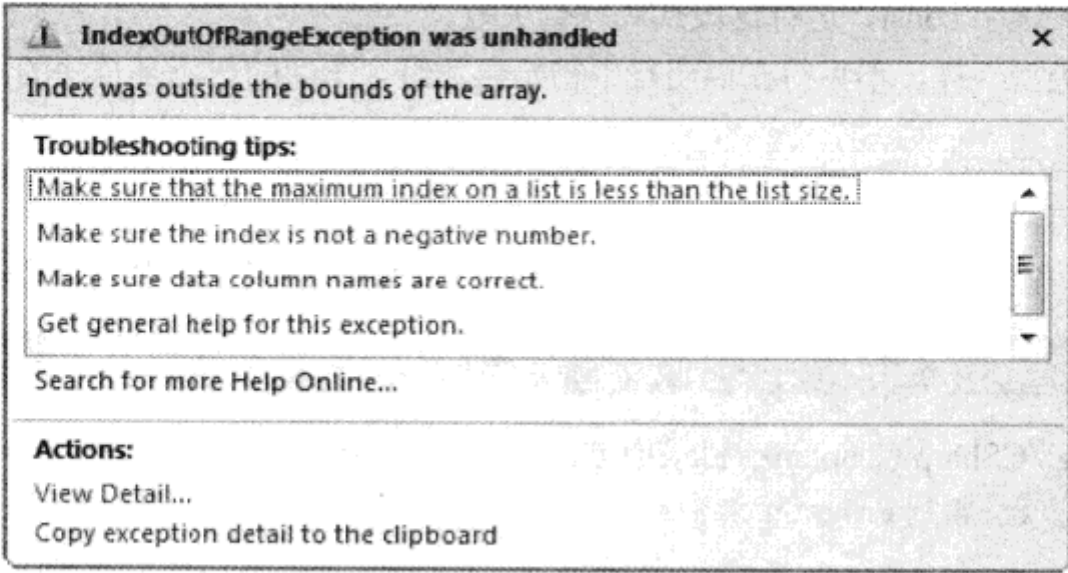


图 5-10

这里代码试图访问 `friendNames[3]`。记住，数组索引从 0 开始，所以最后一个元素是 `friendNames[2]`。如果试图访问超出数组大小的元素，代码就会出问题。还可以通过一个更具弹性的方法来访问数组的所有成员，即使用 `foreach` 循环。

2. foreach 循环

`foreach` 循环可以使用一种简便的语法来定位数组中的每个元素：

```
foreach (<baseType> <name> in <array>)
{
    // can use <name> for each element
}
```

这个循环会迭代每个元素，依次把每个元素放在变量 `<name>` 中，且不存在访问非法元素的危险。

不需要考虑数组中有多少个元素，并确保将在循环中使用每个元素。使用这个循环，可以修改上一个示例中的代码，如下所示：

```
static void Main(string[] args)
{
    string[] friendNames = {"Robert Barwell", "Mike Parry",
                           "Jeremy Beacock"};
    Console.WriteLine("Here are {0} of my friends:",
                      friendNames.Length);
    foreach (string friendName in friendNames)
    {
        Console.WriteLine(friendName);
    }
    Console.ReadKey();
}
```

这段代码的输出结果与前面的示例完全相同。使用这种方法和标准的 for 循环的主要区别在于：foreach 循环对数组内容进行只读访问，所以不能改变任何元素的值。例如，不能编写如下代码：

```
foreach (string friendName in friendNames)
{
    friendName = "Rupert the bear";
}
```

如果编译这段代码，就会失败。但如果使用简单的 for 循环，就可以给数组元素赋值。

### 3. 多维数组

多维数组是使用多个索引访问其元素的数组。例如，假定要确定一座山相对于某位置的高度，可以使用两个坐标 x 和 y 来指定一个位置。把这两个坐标用作索引，数组 hillHeight 就可以用每对坐标来存储高度，这就要使用多维数组了。

像这样的二维数组可以声明如下：

```
<baseType>[,] <name>;
```

多维数组只需要更多的逗号，例如：

```
<baseType>[,,,] <name>;
```

该语句声明了一个 4 维数组。赋值也使用类似的语法，用逗号分隔大小。要声明和初始化二维数组 hillHeight，其基本类型是 double，x 的大小是 3，y 的大小是 4，则需要：

```
double[,] hillHeight = new double[3,4];
```

还可以使用字面值进行初始的赋值。这里使用嵌套的花括号块，用逗号分隔开，例如：

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

这个数组的维度与前面的相同，也是 3 行 4 列。通过提供字面值隐式定义了这些维度。

要访问多维数组中的每个元素，只需指定它们的索引，并用逗号分隔开，例如：

```
hillHeight[2,1]
```

接着就可以像其他元素那样处理它了。这个表达式将访问上面定义的第 3 个嵌套数组中的第 2 个元素(其值是 4)。记住,索引从 0 开始,第一个数字是嵌套的数组。换言之,第一个数字指定花括号对,第 2 个数字指定该对花括号中的元素。用图 5-11 来表示这个数组。

hillHeight [0,0] 1	hillHeight [0,1] 2	hillHeight [0,2] 3	hillHeight [0,3] 4
hillHeight [1,0] 2	hillHeight [1,1] 3	hillHeight [1,2] 4	hillHeight [1,3] 5
hillHeight [2,0] 3	hillHeight [2,1] 4	hillHeight [2,2] 5	hillHeight [2,3] 6

图 5-11

foreach 循环可以访问多维数组中的所有元素,其方式与访问一维数组相同,例如:

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
foreach (double height in hillHeight)
{
    Console.WriteLine("{0}", height);
}
```

元素的输出顺序与赋予字面值的顺序相同(这里显示了元素的标识符,而不是实际值):

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
```

4. 数组的数组

上一节讨论的多维数组可称为矩形数组,这是因为每一行的元素个数都相同。使用上一个示例,任何一个 x 坐标都可以对应 0~3 的 y 坐标。

也可以使用锯齿数组(jagged array),其中每行都有不同的元素个数。为此,需要有这样一个数组,其中的每个元素都是另一个数组。也可以有数组的数组的数组,甚至更复杂的数组。但是,注意这些数组都必须有相同的基本类型。

声明数组的数组,其语法要在数组的声明中指定多个方括号对,例如:

```
int[][] jaggedIntArray;
```

但初始化这样的数组不像初始化多维数组那样简单,例如不能采用以下的声明方式:

```
jaggedIntArray = new int[3][4];
```

即使这样做了,也不是很有效,因为使用简单的多维数组可以较为轻松地获得相同的结果。也

不能使用下面的代码:

```
jaggedIntArray = {{1, 2, 3}, {1}, {1, 2}};
```

有两种方式:可以初始化包含其他数组的数组(为了清晰起见,称之为子数组),然后依次初始化子数组:

```
jaggedIntArray = new int[2][];  
jaggedIntArray[0] = new int[3];  
jaggedIntArray[1] = new int[4];
```

也可以使用上述字面值赋值的一种改进形式:

```
jaggedIntArray = new int[3][] { new int[] { 1, 2, 3 }, new int[] { 1 },  
                                new int[] { 1, 2 } };
```

也可以进行简化,把数组的初始化和声明放在同一行上,如下所示:

```
int[][] jaggedIntArray = { new int[] { 1, 2, 3 }, new int[] { 1 },  
                           new int[] { 1, 2 } };
```

对锯齿数组可以使用 `foreach` 循环,但通常需要使用嵌套方法,才能得到实际数据。例如,假定下述锯齿数组包含 10 个数组,每个数组又包含一个整数数组,其元素是 1~10 的约数:

```
int[][] divisors1To10 = {new int[] {1},  
                          new int[] {1, 2},  
                          new int[] {1, 3},  
                          new int[] {1, 2, 4},  
                          new int[] {1, 5},  
                          new int[] {1, 2, 3, 6},  
                          new int[] {1, 7},  
                          new int[] {1, 2, 4, 8},  
                          new int[] {1, 3, 9},  
                          new int[] {1, 2, 5, 10}};
```

下面的代码会失败:

```
foreach (int divisor in divisors1To10)  
{  
    Console.WriteLine(divisor);  
}
```

这是因为数组 `divisors1To10` 包含 `int[]` 元素,而不是 `int` 元素。必须循环每个子数组和数组本身:

```
foreach (int[] divisorsOfInt in divisors1To10)  
{  
    foreach(int divisor in divisorsOfInt)  
    {  
        Console.WriteLine(divisor);  
    }  
}
```

可以看出,使用锯齿数组的语法要复杂得多!在大多数情况下,使用矩形数组比较简单,这是一种比较简单的存储方式。但是,有时必须使用锯齿数组,且工作效率并不会因此而降低。

## 5.3 字符串的处理

到目前为止，对字符串的使用还仅限于把字符串写到控制台上，从控制台上读取字符串，以及使用+运算符连接字符串。在编写较有趣的应用程序时，会发现字符串的操作非常多。所以，下面用几页的篇幅介绍 C# 中比较常用的字符串处理技巧。

首先要注意，`string` 类型变量可以看作是 `char` 变量的只读数组。这样，就可以使用下面的语法访问每个字符：

```
string myString = "A string";
char myChar = myString[1];
```

但是，不能用这种方式为各个字符赋值。为了获得一个可写的 `char` 数组，可以使用下面的代码，其中使用了数组变量的 `ToCharArray()` 命令：

```
string myString = "A string";
char[] myChars = myString.ToCharArray();
```

接着就可以采用标准方式处理 `char` 数组了。也可以在 `foreach` 循环中使用字符串，例如：

```
foreach (char character in myString)
{
    Console.WriteLine("{0}", character);
}
```

与数组一样，还可以使用 `myString.Length` 获取元素的个数，这将给出字符串中的字符数，例如：

```
string myString = Console.ReadLine();
Console.WriteLine("You typed {0} characters.", myString.Length);
```

其他的基本字符串处理技巧采用与这个 `<string>.ToCharArray()` 命令类似的格式使用命令。两个简单但很有效的命令是 `<string>.ToLower()` 和 `<string>.ToUpper()`。它们可以分别把字符串转换为大写或小写形式。要明白为什么它们非常有用，可以考虑下面的情形：要检查用户的某个响应，例如字符串 `yes`。如果可以把用户输入的字符串转换为小写形式，就也能检查字符串 `YES`、`Yes`、`yeS` 等，第 4 章介绍了这样一个示例：

```
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
{
    // Act on response.
}
```

注意，这个命令与本节的其他命令一样，并没有真正改变应用它的字符串。把这个命令与字符串合并使用，就会创建一个新的字符串，以便与另一个字符串进行比较(如上所述)，或者赋给另一个变量。该变量可能与当前操作的变量相同，例如：

```
userResponse = userResponse.ToLower();
```

这是一个要点，因为只写出下面的代码是没用的：



```
userResponse.ToLower();
```

下面看看在简化用户输入方面还可以做什么。如果用户无意间在输入内容的前面或后面添加了额外的空格，会怎样？此时，上述代码就不起作用了。这就需要删除输入字符串中的空格，此时可以使用`<string>.Trim()`命令来处理。

```
string userResponse = Console.ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
    // Act on response.
}
```

使用该命令，还可以检测如下的字符串：

```
" YES"
"Yes "
```

也可以使用这些命令删除其他字符，只要在一个 `char` 数组中指定这些字符即可，例如：

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
    // Act on response.
}
```

这将从字符串的前面或后面删除所有空格、字母 `e` 和 `s`。如果字符串中没有其他字符，就会检测以下字符串：

```
"Yeeeeees"
" y"
```

还可以使用`<string>.TrimStart()`和`<string>.TrimEnd()`命令。它们可以把字符串的前面或后面的空格删掉。这些命令也需要指定 `char` 数组。

还有另外两个字符串命令可以处理字符串的空格：`<string>.PadLeft()`和`<string>.PadRight()`。它们可以在字符串的左边或右边添加空格，使字符串达到指定的长度。其语法如下：

```
<string>.PadX(<desiredLength>);
```

例如：

```
myString = "Aligned";
myString = myString.PadLeft(10);
```

这将在 `myString` 中把 3 个空格添加到单词 `Aligned` 的左边。这些方法可以用于在列中对齐字符串，特别适用于放置包含数字的字符串。

与修整命令一样，还可以按照第二种方式使用这些命令，即提供要添加到字符串上的字符，这需要一个 `char`，而不是像修整命令那样指定一个 `char` 数组。例如：


```
myString = "Aligned";
myString = myString.PadLeft(10, '-');
```

这将在 myString 的开头加上 3 个短横线。

还有许多这样的字符串处理命令，其中一些只用于非常特殊的情况，在后面的章节中遇到它们时进行讨论。在继续下面内容之前，介绍 Visual C# 2010 Express Edition 和 Visual Studio 2010 中的一个前几章涉及到(特别是本章)的特性。下面的示例会试验语句自动完成功能，IDE 会给出用户可能要插入的代码。

试一试：VS 中的语句自动完成功能

- (1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex05。
- (2) 在 Program.cs 中输入下列代码，注意输入过程中弹出的窗口：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.
}
```

代码段 Ch05Ex05\Program.cs

- (3) 输入最后的句点时，注意会弹出如图 5-12 所示的窗口。



图 5-12

- (4) 不要移动光标，键入 sp，弹出窗口就会改变，显示一个黄色的工具提示窗口，如图 5-13 所示。

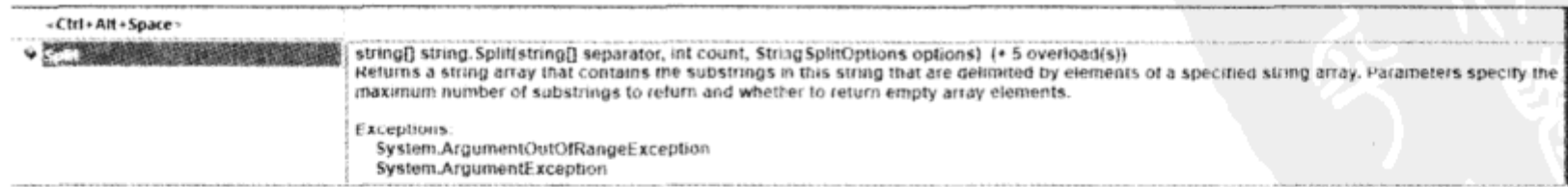


图 5-13

(5) 输入字符 “(se”，会弹出另一个窗口，如图 5-14 所示。

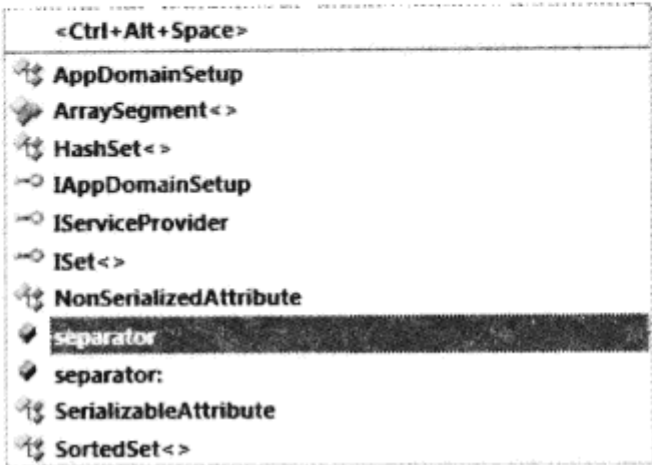


图 5-14

(6) 输入两个字符 “);”，代码如下所示，弹出窗口随之消失。

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = { ' ' };
    string[] myWords;
    myWords = myString.Split(separator);
}
```

(7) 添加下述代码，注意弹出的窗口。

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = { ' ' };
    string[] myWords;
    myWords = myString.Split(separator);
    foreach (string word in myWords)
    {
        Console.WriteLine("{0}", word);
    }
    Console.ReadKey();
}
```

(8) 执行代码，结果如图 5-15 所示。

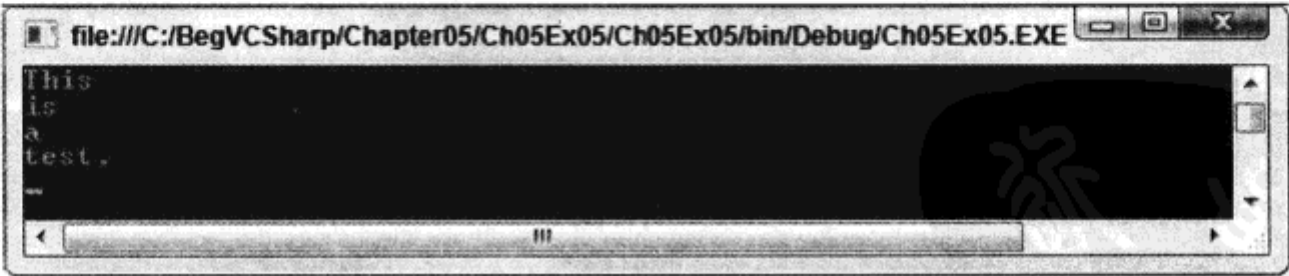


图 5-15

示例的说明

在这段代码中，要注意两点。第一点是所使用的新字符串命令，第二点是使用了自动完成功能。使用命令<string>.Split()把 string 转换为 string 数组，把它在指定的位置分隔开。这些位置采用 char 数组的形式，在本例中该数组只有一个元素，即空格字符：

```
char[] separator = {' '};
```

下面的代码把字符串在每个空格处分解开时，会得到其子字符串，即得到包含单个单词的数组：

```
string[] myWords;  
myWords = myString.Split(separator);
```

接着使用 `foreach` 循环迭代这个数组中的单词，并把这些单词写到控制台上：

```
foreach (string word in myWords)  
{  
    Console.WriteLine("{0}", word);  
}
```



得到的每个单词都没有空格，单词的内部和两端都没有空格。在使用 `split()` 时，删除了分隔符。

接着，看看自动完成功能。VS 和 VCE 都是智能化极高的程序包，在用户键入代码时会提供许多关于代码的信息。即使在一个新行上键入第一个字符，IDE 也试图帮助用户，建议输入关键字、变量名、类型名等。只要在上面的代码中输入 3 个字母 `str`，IDE 就会猜出要输入 `string`。在键入变量名时，这个功能会更有用。在较长的代码中，常常会忘记要使用的变量名。IDE 会在用户键入代码的过程中弹出一系列变量名，用户可以选择，无需查看以前的代码。

在 `myString` 的后面键入句点时，IDE 知道 `myString` 是一个字符串，也知道您要指定一个字符串命令，于是显示可用选项。此时可以停止键入，使用上下箭头键选择需要的命令。在这些可用命令中移动时，IDE 会告诉您当前选中命令的含义，以及使用它的语法。

在开始键入更多字符时，IDE 会把选中的命令移动到命令的顶部，以便自动键入该命令。一旦它显示出需要的命令，就可以继续键入，就像键入完整的名称一样，所以键入 "(" 会直接跳到指定的位置上，在该位置上，有某些命令需要的额外信息——IDE 甚至会告诉用户该信息必须采用的格式，并显示这些接受各种信息的命令选项。

IDE 的这个功能(也称为 `IntelliSense`)使用起来非常方便，可以使您轻松地找到奇怪类型的信息。查看 `string` 类型的所有命令是很有趣的，这不会使计算机崩溃，试一试吧！



有时所显示的信息会遮挡前面已经键入的代码，这是很恼人的。因为在键入时需要引用被遮挡的代码。此时可以按下 `Ctrl` 键，使命令列表变成透明的，以便查看被遮挡的代码。

## 5.4 小结

本章用一定的篇幅扩展了变量的知识。本章最重要的话题是类型转换，后面还要讨论它。掌握本章介绍的概念，会使以后的学习容易得多。

另外还介绍了几个变量类型，它们可以采用更友好的方式存储数据。本章阐述了枚举如何使数值更容易辨识，从而使代码的可读性更高，结构如何在一个地方合并多个相关的数据元素，如何把类似的数据组合到数组中。本书其他地方常用到这些类型。

最后介绍了字符串的处理，讨论了一些基本技巧和规则。C#提供了许多字符串命令，但这里只介绍了其中几个，还说明了如何查看 IDE 中可用的命令。使用这些技巧可以尝试许多工作。至少下面的练习可以使用本章没有讨论的一个或多个字符串命令来完成。

本章扩展了变量的知识，包括：

- 类型转换
- 枚举
- 结构
- 数组
- 字符串处理

## 5.5 练习

(1) 下面的转换哪些不是隐式转换？

- a. int 转换为 short
- b. short 转换为 int
- c. bool 转换为 string
- d. byte 转换为 float

(2) 基于 short 类型的 color 枚举包含彩虹的颜色，再加上黑色和白色，据此编写 color 枚举的代码。这个枚举可以使用 byte 类型吗？

(3) 修改第 4 章的 Mandelbrot 集合生成程序示例，使用下面的结构表示复数：

```
struct imagNum
{
    public double real, imag;
}
```

(4) 下面的代码可以成功编译吗？为什么？

```
string[] blab = new string[5]
string[5] = 5th string.
```

(5) 编写一个控制台应用程序，它接收用户输入的一个字符串，将其中的字符以与输入相反的顺序输出。

(6) 编写一个控制台应用程序，它接收一个字符串，用 yes 替换字符串中所有的 no。

(7) 编写一个控制台应用程序，给字符串中的每个单词加上双引号。

附录 A 给出了练习答案。

5.6 本章要点

主 题	重 要 概 念
类型转换	值可以从一种类型转换为另一种类型，但在转换时应遵循一些规则。隐式转换是自动进行的，但只有源值类型的所有可能值都可以在目标值类型中使用时，才能进行隐式转换。也可以进行显式转换，但可能得不到期望的值，甚至可能出错
枚举	枚举是包含一组离散值的类型，每个离散值都有一个名称。枚举用 <code>enum</code> 关键字定义，以便在代码中理解它们，因为它们的可读性都很高。枚举有基本的数值类型(默认是 <code>int</code> )，可以使用枚举值的这个属性在枚举值和数值之间转换，或者标识枚举值
结构	结构是同时包含几个不同的值的类型。结构用 <code>struct</code> 关键字定义。包含在结构中的每个值都有名称和类型，存储在结构中的每个值的类型不一定相同
数组	数组是同类型数值的集合。数组有固定的大小或长度，确定了数组可以包含多少个值。可以定义多维数组或锯齿数组，来保存不同数量和形状的数据。还可以使用 <code>foreach</code> 循环迭代数组中的值





# 第 6 章

## 函 数

### 本章内容:

- 如何定义和使用不接受或返回任何数据的简单函数
- 如何在函数中传入传出数据
- 使用变量作用域
- 如何结合使用 Main() 函数和命令行参数
- 如何把函数提供为结构类型的成员
- 如何使用函数重载
- 如何使用委托

我们迄今看到的代码都是以单个代码块的形式出现的，其中包含一些重复执行的循环代码，以及有条件地执行的分支语句。如果要对数据执行某种操作，就应把所需要的代码放在合适的地方。

这种代码结构的作用是有限的。某些任务常常需要在一个程序中执行好几次，例如，查找数组中的最大值。此时可以把相同(或几乎相同)的代码块按照需要放在应用程序中，但这样做也会存在问题。在某个常见任务中，即使进行非常小的改动(例如，修改某个代码错误)，也需要修改多个代码块，这些代码块可能分布在整个应用程序中。如果忘了修改其中的一个代码块，就会产生很大的影响，导致整个应用程序失败。另外，应用程序也较长。

解决这个问题的方法是使用函数。在 C# 中，函数是一种方法，可提供在应用程序中的任何一处执行的代码块。



本章介绍的特定类型的函数称为“方法”。但是，这个术语在 .NET 编程中有非常特殊的含义，本书后面会详细讨论它，所以现在不使用这个术语。

例如，有一个函数返回数组中的最大值，可以在代码的任何位置使用这个函数，且在每个地方都使用相同的代码行。因为只需要提供一次这段代码，所以对代码的任何修改将影响使用该函数进

行的计算。这个函数可以看作包含可重用的代码。

函数还可以提高代码的可读性，因为可以使用函数将相关代码组合在一起。这样，应用程序主体就会非常短，因为代码的内部工作被分散了。这类似于在 IDE 中使用大纲视图将代码区域折叠在一起，应用程序的结构更加合理。

函数还可以用于创建多用途的代码，让它们对不同的数据执行相同的操作。可以采用参数形式为函数提供信息，以返回值的形式得到函数的结果。在上面的示例中，参数就是一个要搜索的数组，而返回值就是数组中的最大值。这意味着每次可以使用同一个函数处理不同的数组。函数的名称和参数(不是返回类型)共同定义了函数的签名。

## 6.1 定义和使用函数

本节介绍如何将函数添加到应用程序中，以及如何在代码中使用(调用)它们。首先从基础知识开始，看看不与调用代码交换任何数据的简单函数，然后介绍更高级的函数用法。首先看一个示例。

### 试一试：定义和使用基本函数

(1) 在 C:\BegVCSharp\Chapter06 目录中创建一个新控制台应用程序 Ch06Ex01。

(2) 把下述代码添加到 Program.cs 中：



```
class Program
{
    static void Write()
    {
        Console.WriteLine("Text output from function.");
    }

    static void Main(string[] args)
    {
        Write();
        Console.ReadKey();
    }
}
```

代码段 Ch06Ex01\Program.cs

(3) 执行代码，结果如图 6-1 所示。

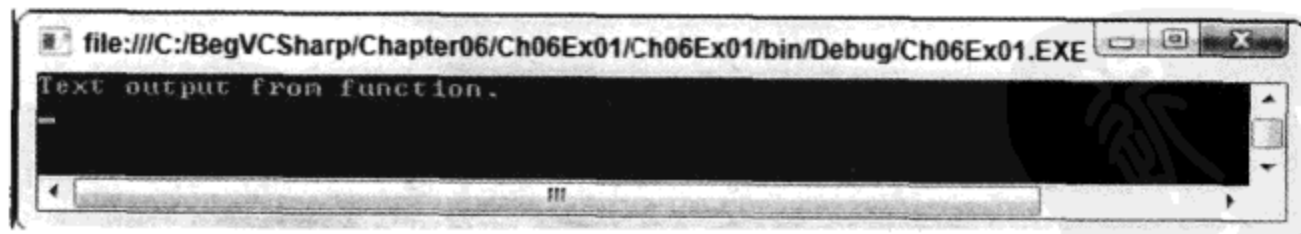


图 6-1

#### 示例的说明

下面的 4 行代码定义了函数 Write()：

```
static void Write()
```

```
{
    Console.WriteLine("Text output from function.");
}
```

这些代码把一些文本输出到控制台窗口中。但此时这些并不重要，我们更关心定义和使用函数的机制。

函数定义由以下几部分组成：

- 两个关键字：static 和 void
- 函数名后跟圆括号，如 Write()
- 一个要执行的代码块，放在花括号中



一般采用 PascalCase 形式编写函数名。

定义 Write()函数的代码非常类似于应用程序中的其他代码：

```
static void Main(string[] args)
{
    ...
}
```

这是因为，到目前为止我们编写的所有代码(除了类型定义之外)都是函数的一部分。函数 Main() 是控制台应用程序的入口点函数。当执行 C#应用程序时，就会调用它包含的入口点函数，这个函数执行完毕后，应用程序就终止了。所有 C#可执行代码都必须有一个入口点。

Main()函数和 Write()函数的唯一区别(除了它们包含的代码)是函数名 Main 后面的圆括号中还有一些代码，这是指定参数的方式，详见后面的内容。

如上所述，Main()函数和 Write()函数都是使用关键字 static 和 void 定义的。关键字 static 与面向对象的概念相关，本书在后面讨论。现在只需记住，在本节的应用程序中所使用的所有函数都必须使用这个关键字。

而 void 更容易解释。这个关键字表明函数没有返回值。本章后面将讨论函数有返回值时需要编写什么代码。

继续下去，调用函数的代码如下所示：

```
Write();
```

键入函数名，后跟空括号即可。在程序执行到这行代码时，就会运行 Write()函数中的代码。



在定义函数和调用函数时，必须使用圆括号。如果删除它们，将无法编译代码。

### 6.1.1 返回值

通过函数进行数据交换的最简单方式是利用返回值。有返回值的函数会计算这个值，其方式与在表达式中使用变量计算它们包含的值完全相同。与变量一样，返回值也有数据类型。

例如，有一个函数 `GetString()`，其返回值是一个字符串，可以在代码中使用该函数，如下所示：

```
string myString;  
myString = GetString();
```

还有一个函数 `GetVal()`，它返回一个 `double` 值，可以在数学表达式中使用它。

```
double myVal;  
double multiplier = 5.3;  
myVal = GetVal() * multiplier;
```

当函数返回一个值时，可以采用以下两种方式修改函数：

- 在函数声明中指定返回值的类型，但不使用关键字 `void`。
- 使用 `return` 关键字结束函数的执行，把返回值传送给调用代码。

从代码角度分析，控制台应用程序函数中的下述代码看起来像是前面见过的函数类型：

```
static <returnType> <functionName>()  
{  
    ...  
    return <returnValue>;  
}
```

这里唯一的限制是 `<returnValue>` 必须是一个值，其类型可以是 `<returnType>`，也可以隐式转换为该类型。但是，`<returnType>` 可以是任何类型，包括前面介绍的较复杂的类型。这段代码可以很简单：

```
static double GetVal()  
{  
    return 3.2;  
}
```

但是，返回值通常是函数执行的一些处理的结果，只需使用 `const` 变量即可得到以上结果。

在执行到 `return` 语句时，程序会立即返回调用代码。这个语句后面的代码都不会执行。但是，这并不意味着 `return` 语句只能放在函数体的最后一行。可以在前边的代码里使用 `return`，也可能在执行了分支逻辑之后使用。把 `return` 语句放在 `for` 循环、`if` 块或其他结构中会使该结构立即终止，函数也立即终止。例如：

```
static double GetVal()  
{  
    double checkVal;  
    // checkVal assigned a value through some logic(not shown here).  
    if (checkVal < 5)  
        return 4.7;  
    return 3.2;  
}
```

根据 `checkVal` 的值，将返回两个值中的一个。这里唯一的限制是 `return` 语句必须在函数的闭合花括号 `}` 之前处理。下面的代码是不合法的：

```
static double GetVal()  
{  
    double checkVal;  
    // checkVal assigned a value through some logic.
```

```

    if (checkVal < 5)
        return 4.7;
}

```

如果 `checkVal >= 5`, 就不会执行到 `return` 语句, 这是不允许的。所有的处理路径都必须执行到 `return` 语句。在大多数情况下, 编译器会检查是否执行到 `return` 语句, 如果没有, 就给出一个错误“并不是所有的处理路径都返回一个值”。

最后需要注意的是, `return` 可以用在通过 `void` 关键字声明的函数中(没有返回值)。如果这么做, 函数就会立即终止。以这种方式使用 `return` 语句时, 在 `return` 关键字和其后的分号之间提供返回值是错误的。

### 6.1.2 参数

当函数接受参数时, 就必须指定下述内容:

- 函数在其定义中指定接受的参数列表, 以及这些参数的类型。
- 在每个函数调用中匹配的参数列表。

这涉及到下述代码:

```

static <returnType> <functionName>(<paramType> <paramName>, ...)
{
    ...
    return <returnValue>;
}

```

其中可以有任意多个参数, 每个参数都有一个类型和一个名称。参数用逗号分隔开。每个参数都在函数的代码中用作一个变量。例如, 下面是一个简单的函数, 带有两个 `double` 参数, 并返回它们的乘积:

```

static double Product(double param1, double param2)
{
    return param1 * param2;
}

```

下面看一个较复杂的示例。

#### 试一试: 通过函数交换数据(1)

- (1) 在 `C:\BegVCSharp\Chapter06` 目录中创建一个新控制台应用程序 `Ch06Ex02`。
- (2) 把下列代码添加到 `Program.cs` 中:



```

class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }
}

```

```

    }

    static void Main(string[] args)
    {
        int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
        int maxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", maxVal);
        Console.ReadKey();
    }
}

```

代码段 Ch06Ex02\Program.cs

(3) 执行代码，结果如图 6-2 所示。

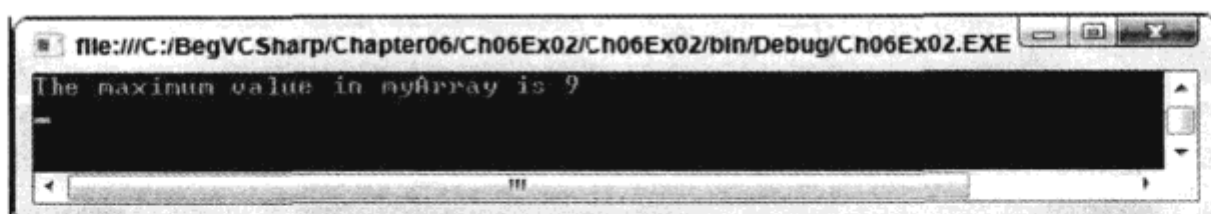


图 6-2

### 示例的说明

这段代码包含一个函数，它执行的任务就是本章引言中示例函数所完成的任务。该函数的参数是一个整数数组，返回该数组中的最大值。该函数的定义如下所示：

```

static int MaxValue(int[] intArray)
{
    int maxVal = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
            maxVal = intArray[i];
    }
    return maxVal;
}

```

函数 `MaxValue()` 定义了一个参数，即 `int` 数组 `intArray`，它还有一个 `int` 类型的返回值。最大值的计算是很简单的。局部整型变量 `maxVal` 初始化为数组中的第一个值，然后把这个值与数组中后面的每个元素依次进行比较。如果一个元素的值比 `maxVal` 大，就用这个值代替当前的 `maxVal` 值。循环结束时，`maxVal` 就包含数组中的最大值，用 `return` 语句返回。

`Main()` 中的代码声明并初始化一个简单的整数数组，用于 `MaxValue()` 函数：

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

调用 `MaxValue()`，把一个值赋给 `int` 变量 `maxVal`：

```
int maxVal = MaxValue(myArray);
```

接着，使用 `Console.WriteLine()` 把这个值写到屏幕上：

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```



## 1. 参数匹配

在调用函数时，必须使参数与函数定义中指定的参数完全匹配，这意味着要匹配参数的类型、个数和顺序。例如，下面的函数：

```
static void MyFunction(string myString, double myDouble)
{
    ...
}
```

不能使用下面的代码调用：

```
MyFunction (2.6, "Hello");
```

这里试图把一个 `double` 值作为第一个参数传递，把 `string` 值作为第二个参数传递，参数的顺序与函数声明中定义的顺序不匹配。

也不能使用下面的代码：

```
MyFunction("Hello");
```

这里仅传送了一个 `string` 参数，而该函数需要两个参数。使用上述两个函数调用都会产生编译错误，因为编译器要求必须匹配所用函数的签名。



使用函数的名称和参数定义函数的签名。

再回顾这个示例，`MaxValue()`只能用于获取整数数组中的最大 `int` 值。如果用下面的代码替换 `Main()`中的代码：

```
static void Main(string[] args)
{
    double[] myArray = {1.3, 8.9, 3.3, 6.5, 2.7, 5.3};
    double maxVal = MaxValue(myArray);
    Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    Console.ReadKey();
}
```

就不能编译这段代码，因为参数类型是错误的。在本章后面的“重载函数”一节将介绍解决这个问题一个有效技术。

## 2. 参数数组

C#允许为函数指定一个(只能指定一个)特定的参数，这个参数必须是函数定义中的最后一个参数，称为参数数组。参数数组可以使用个数不定的参数调用函数，可以使用 `params` 关键字定义它们。

参数数组可以简化代码，因为不必从调用代码中传递数组，而是传递同类型的几个参数，这些参数放在可在函数中使用的一个数组中。

定义使用参数数组的函数时，需要使用下列代码：

```
static <returnType> <functionName>(<p1Type> <p1Name>, ... ,
```

```

                                params <type>[] <name>)
{
    ...
    return <returnValue>;
}

```

使用下面的代码可以调用该函数。

```
<functionName>(<p1>, ... , <val1>, <val2>, ...)
```

其中<val1>和<val2>等都是<type>类型的值，用于初始化<name>数组。可以指定的参数个数几乎不受限制。唯一的限制是它们都必须是<type>类型。甚至可以根本不指定参数。

这一点使参数数组特别适合于为在处理过程中要使用的函数指定其他信息。例如，假定有一个函数 `GetWord()`，它的第一个参数是一个 `string` 值，并返回字符串中的第一个单词。

```
string firstWord = GetWord("This is a sentence.");
```

其中 `firstWord` 被赋予字符串 `This`。

可在 `GetWord()` 中添加一个 `params` 参数，以根据其索引选择另一个要返回的单词：

```
string firstWord = GetWord("This is a sentence.", 2);
```

假定第一个单词计数为 1，则 `firstWord` 就被赋予字符串 `is`。

也可以在第 3 个参数中限制返回的字符个数，同样通过 `params` 参数来实现：

```
string firstWord = GetWord("This is a sentence.", 4, 3);
```

此时 `firstWord` 被赋予字符串 `sen`。

下面的示例定义并使用带有 `params` 类型参数的函数。

### 试一试：通过函数交换数据(2)

- (1) 在 `C:\BegVCSharp\Chapter06` 目录中创建一个控制台应用程序 `Ch06Ex03`。
- (2) 把下述代码添加到 `Program.cs` 中：



```

class Program
{
    static int SumVals(params int[] vals)
    {
        int sum = 0;
        foreach (int val in vals)
        {
            sum += val;
        }
        return sum;
    }

    static void Main(string[] args)
    {
        int sum = SumVals(1, 5, 2, 9, 8);
        Console.WriteLine("Summed Values = {0}", sum);
    }
}

```

```

        Console.ReadKey();
    }
}

```

代码段 Ch06Ex03\Program.cs

(3) 执行代码，结果如图 6-3 所示。

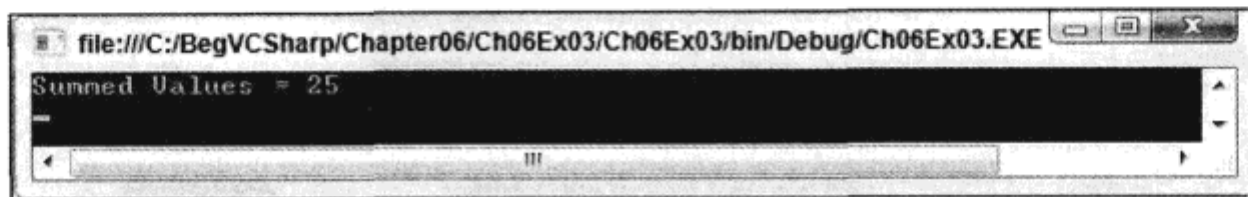


图 6-3

#### 示例的说明

这个示例用关键字 `params` 定义函数 `sumVals()`，该函数可以接受任意个 `int` 参数(但不接受其他类型的参数)：

```

static int SumVals(params int[] vals)
{
    ...
}

```

这个函数对 `vals` 数组中的值进行迭代，把这些值加在一起，返回其结果。

在 `Main()` 中，用 5 个整型参数调用函数 `SumVals()`：

```
int sum = SumVals(1, 5, 2, 9, 8);
```

也可以用 0、1、2 或 100 个整型参数调用这个函数——参数的数量不受限制。

### 3. 引用参数和值参数

本章迄今定义的所有函数都带有值参数。其含义是，在使用参数时，是把一个值传递给函数使用的一个变量。对函数中此变量的任何修改都不影响函数调用中指定的参数。例如，下面的函数使传递过来的参数值加倍，并显示出来：

```

static void ShowDouble(int val)
{
    val *= 2;
    Console.WriteLine("val doubled = {0}", val);
}

```

参数 `val` 在这个函数中被加倍，如果按以下方式调用它：

```

int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);

```

输出到控制台上的文本如下所示：

```
myNumber = 5
```

```
val doubled = 10
myNumber = 5
```

把 `myNumber` 作为一个参数, 调用 `ShowDouble()` 并不影响 `Main()` 中 `myNumber` 的值, 即使赋予 `val` 的参数被加倍, `myNumber` 的值也不变。

这很不错, 但如果要改变 `myNumber` 的值, 就会有问题。可以使用一个为 `myNumber` 返回新值的函数:

```
static int DoubleNum(int val)
{
    val *= 2;
    return val;
}
```

并使用下面的代码调用它:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = DoubleNum(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

但这段代码一点也不直观, 且不能改变用作参数的多个变量值(因为函数只有一个返回值)。

此时可以通过“引用”传递参数。即函数处理的变量与函数调用中使用的变量相同, 而不仅仅是值相同的变量。因此, 对这个变量进行的任何改变都会影响用作参数的变量值。为此, 只需使用 `ref` 关键字指定参数:

```
static void ShowDouble(ref int val)
{
    val *= 2;
    Console.WriteLine("val doubled = {0}", val);
}
```

在函数调用中(这是必须的, 因为 `ref` 参数是函数签名的一部分)再次指定它:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

输出到控制台的文本如下所示:

```
myNumber = 5
val doubled = 10
myNumber = 10
```

这次, `myNumber` 被 `ShowDouble()` 修改了。

用作 `ref` 参数的变量有两个限制。首先, 函数可能会改变引用参数的值, 所以必须在函数调用中使用“非常量”变量。所以, 下面的代码是非法的:

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

其次，必须使用初始化过的变量。C#不允许假定 `ref` 参数在使用它的函数中初始化，下面的代码也是非法的：

```
int myNumber;
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

#### 4. 输出参数

除了按引用传递值之外，还可以使用 `out` 关键字，指定所给的参数是一个输出参数。`out` 关键字的使用方式与 `ref` 关键字相同(在函数定义和函数调用中用作参数的修饰符)。实际上，它的执行方式与引用参数完全一样，因为在函数执行完毕后，该参数的值将返回给函数调用中使用的变量。但是，存在一些重要区别。

- 把未赋值的变量用作 `ref` 参数是非法的，但可以把未赋值的变量用作 `out` 参数。
- 另外，在函数使用 `out` 参数时，`out` 参数必须看作是还未赋值。

即调用代码可以把已赋值的变量用作 `out` 参数，存储在该变量中的值会在函数执行时丢失。

例如，考虑前面返回数组中最大值的 `MaxValue()` 函数，略微修改该函数，获取数组中最大值的元素索引。为简单起见，如果数组中有多个元素的值都是这个最大值，只提取第一个最大值的索引。为此，修改函数，添加一个 `out` 参数，如下所示：

```
static int MaxValue(int[] intArray, out int maxIndex)
{
    int maxVal = intArray[0];
    maxIndex = 0;
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
        {
            maxVal = intArray[i];
            maxIndex = i;
        }
    }
    return maxVal;
}
```

可以采用以下方式使用该函数：

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine("The maximum value in myArray is {0}",
    MaxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}",
    maxIndex + 1);
```

结果如下：

```
The maximum value in myArray is 9
The first occurrence of this value is at element 7
```

注意，必须在函数调用中使用 `out` 关键字，就像 `ref` 关键字一样。



在屏幕上显示结果时，给返回的 `maxIndex` 的值加上 1。这样可以使索引更容易读懂，因此数组的第一个元素称为元素 1，而不是元素 0。

## 6.2 变量的作用域

在上一节中，读者可能想知道为什么需要利用函数交换数据。原因是 C# 中的变量仅能从代码的本地作用域访问。给定的变量有一个作用域，访问该变量要通过这个作用域来实现。

变量的作用域是一个重要的主题，最好用一个示例加以说明。下面的示例将演示变量在一个作用域中定义，但试图在另一个作用域中使用的情形。

### 试一试：变量的作用域

(1) 对 Ch06Ex01 中的 Program.cs 进行如下修改：



可从  
wrox.com  
下载源代码

```
class Program
{
    static void Write()
    {
        Console.WriteLine("myString = {0}", myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
        Console.ReadKey();
    }
}
```

代码段 Ch06Ex01\Program.cs

(2) 编译代码，注意显示在任务列表中的错误和警告：

```
The name 'myString' does not exist in the current context
The variable 'myString' is assigned but its value is never used
```

### 示例的说明

什么地方出错了？不能在 `Write()` 函数中访问在应用程序主体(`Main()`函数)中定义的变量 `myString`。原因是变量是有作用域的，在这个作用域中，变量才是有效的。这个作用域包括定义变量的代码块和直接嵌套在其中的代码块。函数中的代码块与调用它们的代码块是不同的。在 `Write()` 中，没有定义 `myString`，在 `Main()` 中定义的 `myString` 则超出了作用域——它只能在 `Main()` 中使用。

实际上，在 `Write()` 中可以有一个完全独立的变量 `myString`，修改代码，如下所示：

```
class Program
{
```



```

static void Write()
{
    string myString = "String defined in Write()";
    Console.WriteLine("Now in Write()");
    Console.WriteLine("myString = {0}", myString);
}

static void Main(string[] args)
{
    string myString = "String defined in Main()";
    Write();
    Console.WriteLine("\nNow in Main()");
    Console.WriteLine("myString = {0}", myString);
    Console.ReadKey();
}
}

```

这段代码就可以编译，输出结果如图 6-4 所示。

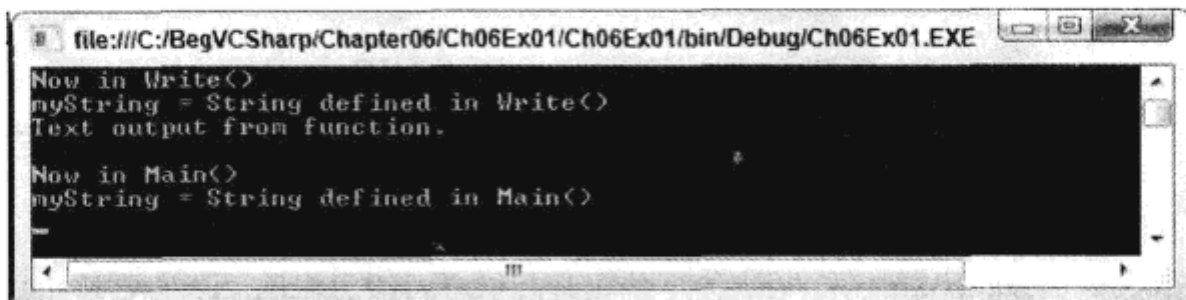


图 6-4

这段代码执行的操作如下：

- Main()定义和初始化字符串变量 myString。
- Main() 把控制权传送给 Write()。
- Write()定义和初始化字符串变量 myString，它与 Main()中定义的 myString 变量完全不同。
- Write()把一个字符串输出到控制台上，该字符串包含在 Write()中定义的 myString 的值。
- Write()把控制权传送回 Main()。
- Main()把一个字符串输出到控制台上，该字符串包含在 Main()中定义的 myString 的值。

其作用域以这种方式覆盖一个函数的变量称为局部变量。还有一种全局变量，其作用域可覆盖多个函数。修改代码，如下所示：

```

class Program
{
    static string myString;

    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Program.myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
    }
}

```

```

    Program.myString = "Global string";
    Write();
    Console.WriteLine("\nNow in Main()");
    Console.WriteLine("Local myString = {0}", myString);
    Console.WriteLine("Global myString = {0}", Program.myString);
    Console.ReadKey();
}
}

```

结果如图 6-5 所示。

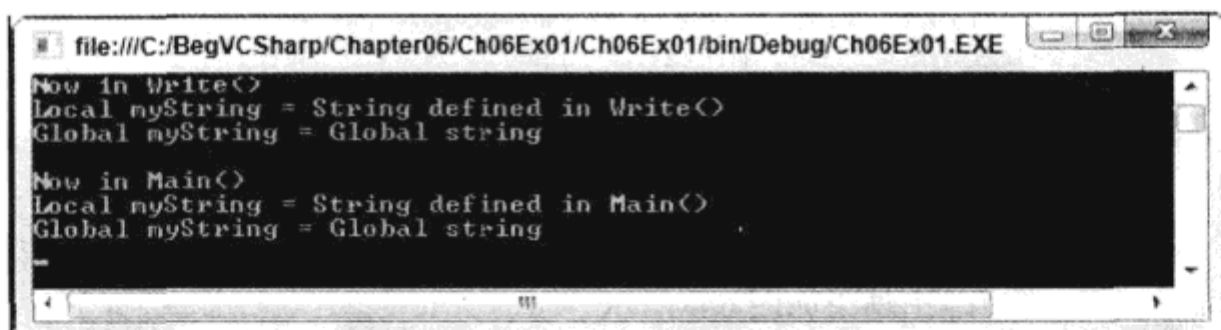


图 6-5

这里添加了另一个变量 `myString`，这次进一步加深了代码中的名称层次。这个变量定义如下：

```
static string myString;
```

注意，这里也需要 `static` 关键字。在这种类型的控制台应用程序中，必须使用 `static` 或 `const` 关键字，来定义这种形式的全局变量。如果要修改全局变量的值，就需要使用 `static`，因为 `const` 禁止修改变量的值。

为了区分这个变量和 `Main()` 与 `Write()` 中的同名局部变量，必须用一个完整限定的名称为变量名分类，参见第 3 章。这里把全局变量称为 `Program.myString`。注意，在全局变量和局部变量同名时，这是必需的。如果没有局部 `myString` 变量，就可以使用 `myString` 表示全局变量，而不需要使用 `Program.myString`。如果局部变量和全局变量同名，全局变量就会被屏蔽。

全局变量的值在 `Main()` 中设置如下：

```
Program.myString = "Global string";
```

在 `Write()` 中可以通过如下语句访问：

```
Console.WriteLine("Global myString = {0}", Program.myString);
```

为什么不能使用这个技术通过函数交换数据，而要使用前面介绍的参数来交换数据？有时，这确实是一种交换数据的首选方式，但在许多情况下不应使用这种方式。是否使用全局变量取决于函数的位置。使用全局变量的问题在于，它们通常不适合于“常规用途”的函数——这些函数能处理我们所提供的任意数据，而不仅限于处理特定全局变量中的数据。详见本章后面的内容。

### 6.2.1 其他结构中变量的作用域


上一节的一个要点总结了上述内容，并超出了函数之间的变量作用域。前面说过，变量的作用域包含定义它们的代码块和直接嵌套在其中的代码块。这也可以应用到其他代码块上，例如分支和循环结构的代码块。考虑下面的代码：

```
int i;
for (i = 0; i < 10; i++)
{
    string text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

字符串变量 `text` 是 `for` 循环的局部变量，这段代码不能编译，因为在该循环外部调用的 `Console.WriteLine()` 试图使用该变量 `text`，这超出了循环的作用域。修改代码，如下所示：

```
int i;
string text;
for (i = 0; i < 10; i++)
{
    text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

这段代码也会失败，原因是必须在使用变量前对其进行声明和初始化，而 `text` 是在 `for` 循环中初始化的。赋给 `text` 的值在循环块退出时就丢失了。但是还可以进行如下修改：


[wrox.com](http://wrox.com) 可从  
 下载源代码

```
int i;
string text = "";
for (i = 0; i < 10; i++)
{
    text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

代码段 VariableScopeInLoops\Program.cs

这次 `text` 是在循环外部初始化的，可以访问它的值。这段简单代码的结果如图 6-6 所示。

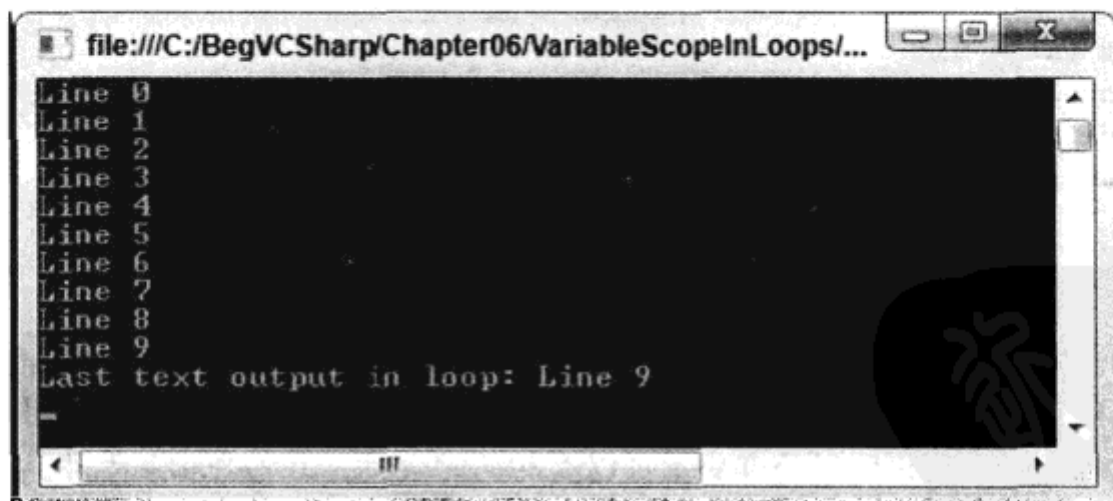


图 6-6

在循环中最后赋给 `text` 的值可以在循环外部访问。可以看出，这个主题的内容需要花一点时间来掌握。在前面的示例中，循环之前赋给 `text` 空字符串，而在循环之后的代码中，该 `text` 就不会是空字符串了，其原因并不明显。

这种情况的解释涉及到分配给 `text` 变量的内存空间，实际上任何变量都是这样。只声明一个简单变量类型，并不会引起其他的变化。只有在给变量赋值后，这个值才占用一块内存空间。如果这种占据内存空间的行为在循环中发生，该值实际上定义为一个局部值，在循环的外部会超出了其作用域。

即使变量本身没有局部化到循环上，循环所包含的值也局部化到该循环上。但是，在循环外部赋值可以确保该值是主体代码的局部值，在循环内部它仍处于其作用域中。这意味着变量在退出主体代码块之前是没有超出作用域的，所以可以在循环外部访问它的值。

幸好，C#编译器可检测变量作用域的问题，它生成的响应错误信息有助于我们理解变量作用域问题。

最后一个要注意的问题是，应采用“最佳实践方式”。一般情况下，最好在声明和初始化所有变量后，再在代码块中使用它们。一个例外是把循环变量声明为循环块的一部分，例如：

```
for (int i = 0; i < 10; i++)
{
    ...
}
```

其中 `i` 局部化于循环代码块中，但这是可以的，因为很少需要在外部的代码中访问这个计数器。

## 6.2.2 参数和返回值与全局数据

本节将详细介绍如何通过全局数据以及参数和返回值与函数交换数据。先看看下面的代码：

```
class Program
{
    static void ShowDouble(ref int val)
    {
        val *= 2;
        Console.WriteLine("val doubled = {0}", val);
    }

    static void Main(string[] args)
    {
        int val = 5;
        Console.WriteLine("val = {0}", val);
        ShowDouble(ref val);
        Console.WriteLine("val = {0}", val);
    }
}
```



这段代码与本章前面的代码稍有不同，在前面的示例中，在 `Main()` 中使用了变量名 `myNumber`，这说明了局部变量可以有相同的名称，且不会相互干涉。这里列出的两个代码示例更加类似，以便我们集中精力研究它们的区别，而无需担心变量名。

和下面的代码比较：

```

class Program
{
    static int val;

    static void ShowDouble()
    {
        val *= 2;
        Console.WriteLine("val doubled = {0}", val);
    }

    static void Main(string[] args)
    {
        val = 5;
        Console.WriteLine("val = {0}", val);
        ShowDouble();
        Console.WriteLine("val = {0}", val);
    }
}

```

这两个 showDouble()函数的结果是相同的。

使用哪种方法并没有什么硬性规定，这两种方法都十分有效。但是，需要考虑一些规则。

首先，在第一次讨论这个问题时，使用全局值的 showDouble()版本只使用全局变量 val。为了使用这个版本，就必须使用这个全局变量。这会对该函数的多样性有轻微的限制，如果要存储结果，就必须总是把这个全局变量值复制到其他变量中。另外，全局数据可以在应用程序的其他地方由代码修改，这会导致预料不到的结果(其值可能会改变，等我们没有认识到这一点时为时已晚)。

但是，损失了多样性常常是有好处的。我们常常希望把一个函数只用于一个目的，使用全局数据存储能减少在函数调用中犯错的可能性，例如把它传递给错误的变量。

当然，也可以说，这种简化实际上使代码更难理解。显式指定参数可以一眼看出发生了什么改变。如 FunctionName(val1, out val2)函数调用，其中 val1 和 val2 都是要考虑的重要变量，在函数执行结束后，val2 就会被赋予一个新值。反之，如果这个函数不带参数，就不能对它处理了什么数据做任何假设。

最后，记住未必总是能使用全局数据。本书的后面将介绍在不同的文件中编写的代码，以及不同名称空间中的代码如何通过函数彼此通信。像这样的情况，代码常常要分开编写，显然不能使用全局存储方式。

总之，可以自由选择使用哪种技术来交换数据。一般情况下，最好使用参数，而不使用全局数据，但有时使用全局数据更合适，使用这种技术并没有错。

## 6.3 Main()函数

前面介绍了创建和使用函数时涉及的大多数简单技术，下面详细论述 Main()函数。

Main()是 C#应用程序的入口点，执行这个函数就是执行应用程序。也就是说，在执行过程开始时，会执行 Main()函数，在 Main()函数执行完毕时，执行过程就结束了。

这个函数可以返回 void 或 int，有一个可选参数 string[] args。Main()函数可以使用如下 4 种版本：

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

上面的第三、四个版本返回一个 `int` 值，它们可以用于表示应用程序如何终止，通常用作一种错误提示(但这不是强制的)，一般情况下，返回 0 反映了“正常”的终止(即应用程序执行完毕，并安全地终止)。

`Main()` 的可选参数 `args` 是从应用程序的外部接受信息的方法，这些信息在运行期间指定，其形式是命令行参数。

前面已经遇到了命令行参数，在命令行上执行应用程序时，通常可以直接指定信息，如在执行应用程序时加载一个文件。例如，考虑 Windows 中的记事本应用程序。在命令提示窗口中键入 `notepad`，或者在 Windows 的 Start 菜单中选择 Run 选项，再在打开的窗口中键入 `notepad`，就可以运行该应用程序。也可以键入 `notepad "myfile.txt"`，结果是 Notepad 在运行时将加载文件 `myfile.txt`，如果该文件不存在，Notepad 也会创建该文件。这里 `myfile.txt` 是一个命令行参数。利用 `args` 参数，可以编写以类似方式工作的控制台应用程序。

在执行控制台应用程序时，指定的任何命令行参数都放在这个 `args` 数组中，接着可以根据需要在应用程序中使用这些参数。下面用一个示例来说明。这个示例可以指定任意数量的命令行参数，每个参数都输出到控制台上。

#### 试一试：命令行参数

- (1) 在 `C:\BegVCSharp\Chapter06` 目录中创建一个新控制台应用程序 `Ch06Ex04`。
- (2) 把下列代码添加到 `Program.cs` 中：



```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("{0} command line arguments were specified:",
                           args.Length);
        foreach (string arg in args)
            Console.WriteLine(arg);
        Console.ReadKey();
    }
}
```

代码段 Ch06Ex04\Program.cs

- (3) 打开项目的属性页面(在 Solution Explorer 窗口中右击 `Ch06Ex04` 项目名称，选择 Properties 选项)。
- (4) 选择 Debug 页面，在 Command Line Arguments 设置中添加所希望的命令行参数，如图 6-7 所示。



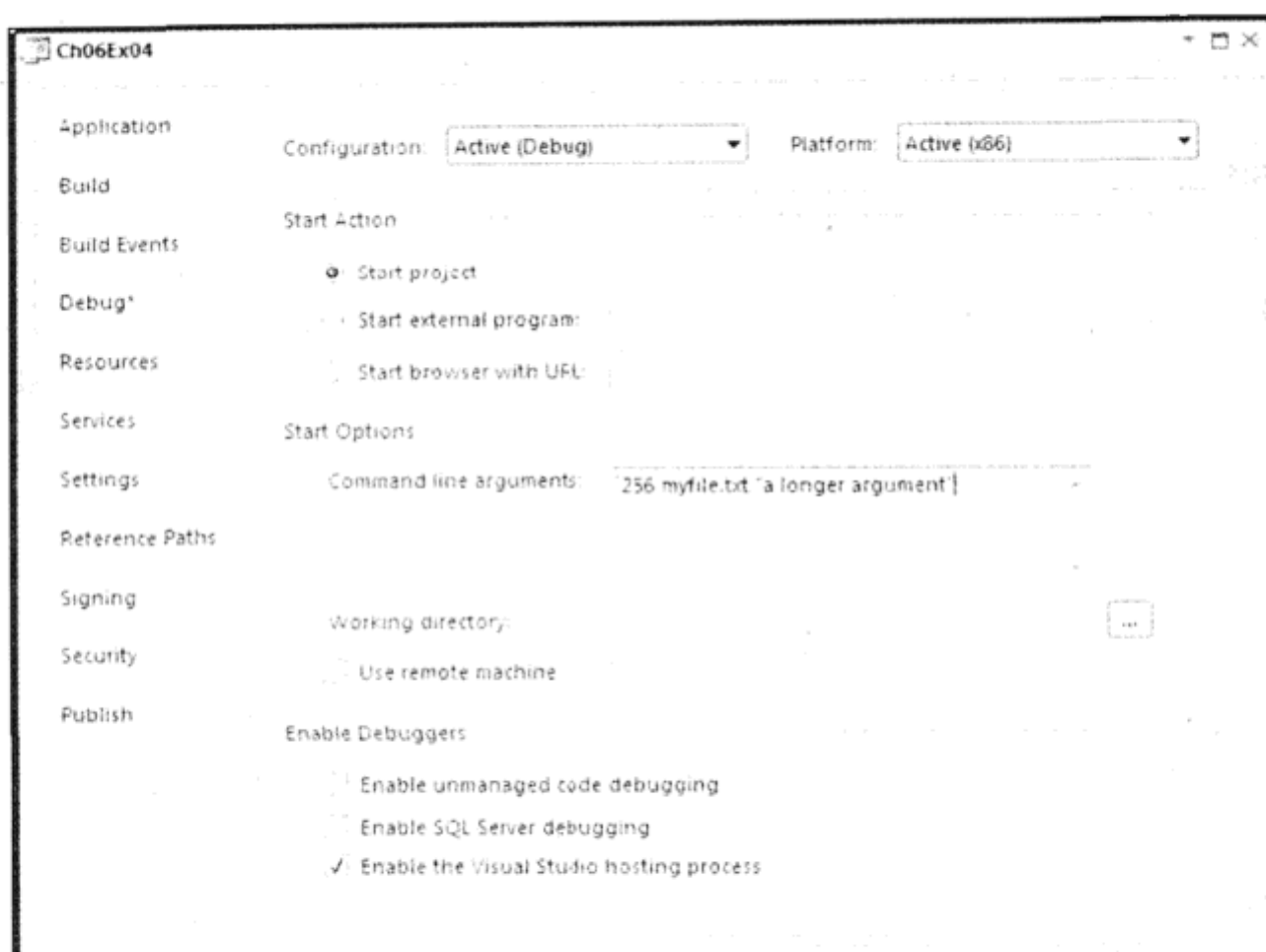


图 6-7

(5) 运行应用程序，输出结果如图 6-8 所示。

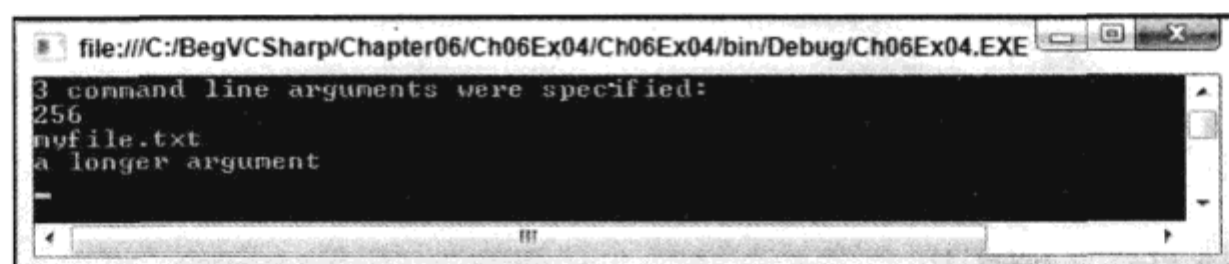


图 6-8

### 示例的说明

这里使用的代码非常简单：

```
Console.WriteLine("{0} command line arguments were specified:",
                  args.Length);
foreach (string arg in args)
    Console.WriteLine(arg);
```

使用 `args` 参数与使用其他字符串数组类似。我们没有对参数进行任何异样的操作，只是把指定的信息写到屏幕上。在本示例中，通过 IDE 中的项目属性提供参数，这是一种很便捷的方式，只要在 IDE 中运行应用程序，就可以使用相同的命令行参数，无需每次都在命令行提示窗口中键入它们。在项目输出所在的目录(C:\BegVSharp\Chapter06\Ch06Ex04\bin\Debug)下打开命令提示窗口，键入下述代码，也可以得到同样的结果：

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

每个参数都用空格分开，如果参数包含空格，就可以用双引号把参数括起来，这样才不会把这

个参数解释为多个参数。

## 6.4 结构函数

第 5 章介绍了结构类型，它可在一个地方存储多个数据元素，结构可以做的工作远不止此。一个重要的功能就是结构可以包含函数和数据。这初看起来很奇怪，但实际上是非常有用的。例如，考虑以下结构：

```
struct customerName
{
    public string firstName, lastName;
}
```

如果变量的类型是 `customerName`，并且要在控制台上输出一个完整的名称，就必须从其组件部分建立该名称。例如，`customerName` 变量 `myCustomer` 可以使用下述语法：

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine("{0} {1}", myCustomer.firstName, myCustomer.lastName);
```

把函数添加到结构中，就可以集中处理常见任务，从而简化这个过程。可以把合适的函数添加到结构类型中，如下所示：

```
struct customerName
{
    public string firstName, lastName;

    public string Name ()
    {
        return firstName + " " + lastName;
    }
}
```

看起来这与本章前面的其他函数很类似，但没有使用 `static` 修饰符。本书将在后面阐明其原因，现在知道该关键字不是结构函数所必须的即可。这个函数的用法如下所示：

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(myCustomer.Name());
```

这个语法比前面的语法简单得多，也更容易理解。注意，`Name()` 函数可以直接访问 `firstName` 和 `lastName` 结构成员，在 `customerName` 结构中，它们可以被看作是全局成员。

## 6.5 函数的重载

本章的前面介绍了在调用函数时，必须匹配函数的签名。这表明，需要让多个函数操作不同类

型的变量。函数重载允许创建多个同名函数，这些函数可使用不同的参数类型。例如，前面使用了下述代码，其中包含函数 `MaxValue()`：

```
class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }

    static void Main(string[] args)
    {
        int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
        int maxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", maxVal);
        Console.ReadKey();
    }
}
```

这个函数只能用于处理 `int` 数组，现在要为不同的参数类型提供不同名称的函数，可以把上述函数重命名为 `IntArrayMaxValue()`，添加诸如 `DoubleArrayMaxValue()` 的函数处理其他类型。另外，还可以在代码中添加如下函数：

```
...
static double MaxValue(double[] doubleArray)
{
    double maxVal = doubleArray[0];
    for (int i = 1; i < doubleArray.Length; i++)
    {
        if (doubleArray[i] > maxVal)
            maxVal = doubleArray[i];
    }
    return maxVal;
}
...
```

这里的区别是使用了 `double` 值。函数名称 `MaxValue()` 是相同的，但其签名是不同的。这是因为如前所述，函数的签名包含函数的名称及其参数。用相同的签名定义两个函数是错误的，但因为这两个函数的签名不同，所以是可行的。



函数的返回类型不是其签名的一部分，所以不能定义两个仅返回类型不同的函数，它们实际上有相同的签名。

添加了前面的代码后，现在有两个版本的 `MaxValue()`，它们的参数是 `int` 和 `double` 数组，分别

返回 `int` 或 `double` 最大值。

这种代码的优点是不必显式地指定要使用哪个函数。只需提供一个数组参数，就可以根据使用的参数类型执行相应的函数。

此时，应注意 VS 和 VCE 中 `IntelliSense` 的另一项功能。如果在应用程序中有上述两个函数，而且要在 `Main()` 中键入函数的名称，IDE 就可以显示出可用的重载函数。如果键入下面的代码：

```
double result = MaxValue(
```

IDE 就会提供两个 `MaxValue()` 版本的信息，使用上下箭头键在其间滚动，如图 6-9 所示。

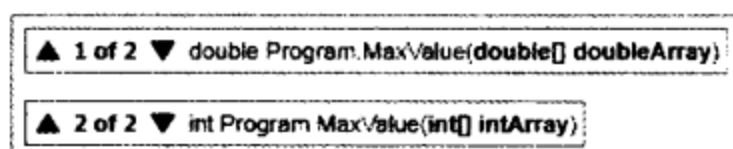


图 6-9

在重载函数时，应包括函数签名的所有方面。例如，有两个不同的函数，它们分别带有值参数和引用参数：

```
static void ShowDouble(ref int val)
{
    ...
}

static void ShowDouble(int val)
{
    ...
}
```

选择使用哪个版本纯粹是根据函数调用是否包含 `ref` 关键字来确定的。下面的代码将调用引用版本：

```
ShowDouble(ref val);
```

下面的代码是调用值版本：

```
ShowDouble(val);
```

另外，还可以根据参数的个数等来区分函数。

## 6.6 委托

委托(delegate)是一种可以把引用存储为函数的类型。这听起来相当棘手，但其机制是非常简单的。委托最重要的用途在本书后面介绍到事件和事件处理时才能解释清楚，但这里也将介绍有关委托的许多内容。委托的声明非常类似于函数，但不带函数体，且要使用 `delegate` 关键字。委托的声明指定了一个返回类型和一个参数列表。

在定义了委托后，就可以声明该委托类型的变量。接着把这个变量初始化为与委托有相同返回类型和参数列表的函数引用。之后，就可以使用委托变量调用这个函数，就像该变量是一个函数一样。

有了引用函数的变量后，还可以执行不能用其他方式完成的操作。例如，可以把委托变量作为参数传递给一个函数，这样，该函数就可以使用委托调用它引用的任何函数，而且在运行之前无需知道调用的是哪个函数。下面的示例使用委托访问两个函数中的一个。

### 试一试：使用委托来调用函数

- (1) 在 C:\BegVCSharp\Chapter06 目录中创建一个新控制台应用程序 Ch06Ex05。
- (2) 把下列代码添加到 Program.cs 中：



```
class Program
{
    delegate double ProcessDelegate(double param1, double param2);

    static double Multiply(double param1, double param2)
    {
        return param1 * param2;
    }
    static double Divide(double param1, double param2)
    {
        return param1 / param2;
    }

    static void Main(string[] args)
    {
        ProcessDelegate process;
        Console.WriteLine("Enter 2 numbers separated with a comma:");
        string input = Console.ReadLine();
        int commaPos = input.IndexOf(',');
        double param1 = Convert.ToDouble(input.Substring(0, commaPos));
        double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
            input.Length - commaPos - 1));
        Console.WriteLine("Enter M to multiply or D to divide:");
        input = Console.ReadLine();
        if (input == "M")
            process = new ProcessDelegate(Multiply);
        else
            process = new ProcessDelegate(Divide);
        Console.WriteLine("Result: {0}", process(param1, param2));
        Console.ReadKey();
    }
}
```

代码段 Ch06Ex05\Program.cs

- (3) 执行代码，结果如图 6-10 所示。

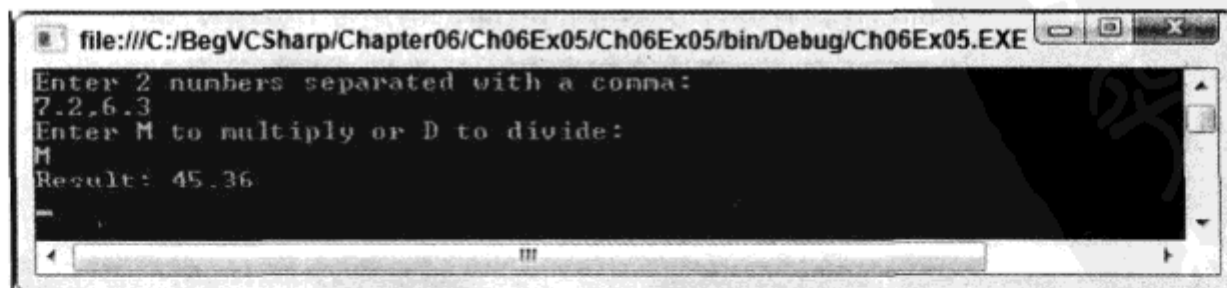


图 6-10

### 示例的说明

这段代码定义了一个委托 `ProcessDelegate`，其返回类型和参数与函数 `Multiply()` 和 `Divide()` 相匹配。委托的定义如下所示：

```
delegate double ProcessDelegate(double param1, double param2);
```

`delegate` 关键字指定该定义是用于委托的，而不是用于函数的(该定义所在的位置与函数定义相同)。接着，该定义指定 `double` 返回类型和两个 `double` 参数。实际使用的名称可以是任意的，所以可以给委托类型和参数指定任意名称。这里委托名是 `ProcessDelegate`，`double` 参数名是 `param1` 和 `param2`。

`Main()` 中的代码首先使用新的委托类型声明一个变量：

```
static void Main(string[] args)
{
    ProcessDelegate process;
```

接着用一些比较标准的 C# 代码请求由逗号分隔的两个数字，并把这些数字放在两个 `double` 变量中：

```
Console.WriteLine("Enter 2 numbers separated with a comma:");
string input = Console.ReadLine();
int commaPos = input.IndexOf(',');
double param1 = Convert.ToDouble(input.Substring(0, commaPos));
double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
    input.Length -- commaPos -- 1));
```



为了说明问题，这里没有验证用户输入的有效性。如果这些是“现实中的”代码，就应花更多的时间来确保在局部变量 `param1` 和 `param2` 中得到有效的值。

接着，询问用户是要相乘，还是相除这两个数字：

```
Console.WriteLine("Enter M to multiply or D to divide:");
input = Console.ReadLine();
```

根据用户的选择，初始化 `process` 委托变量：

```
if (input == "M")
    process = new ProcessDelegate(Multiply);
else
    process = new ProcessDelegate(Divide);
```

要把一个函数引用赋给委托变量，需要使用略显古怪的语法。这个过程比较类似于给数组赋值，必须使用 `new` 关键字创建一个新委托。在这个关键字的后面，指定委托类型，提供一个引用所需函数的参数，该函数是 `Multiply()` 或 `Divide()`。注意这个参数与委托类型或目标函数的参数不匹配，这是委托赋值的一个独特语法，参数是要使用的函数名，且不带括号。

实际上，这里可以使用略微简单的语法：

```
if (input == "M")
    process = Multiply;
```



```
else
    process = Divide;
```

编译器会发现, `process` 变量的委托类型匹配两个函数的签名, 于是自动初始化一个委托。可以自行确定使用哪个语法, 但一些人喜欢使用较长的版本, 因为它更容易一眼看出会发生什么。

最后, 使用该委托调用所选的函数。无论委托引用的是什么函数, 该语法都是有效的:

```
Console.WriteLine("Result: {0}", process(param1, param2));
Console.ReadKey();
}
```

这里把委托变量看作一个函数名。但与函数不同, 我们还可以对这个变量执行更多的操作, 例如, 通过参数将其传递给一个函数, 这个函数的一个简单示例如下:

```
static void ExecuteFunction(ProcessDelegate process)
{
    process(2.2, 3.3);
}
```

就像选择一个要使用的“插件”一样, 把它们传递给函数委托, 就可以控制函数的执行。例如, 一个函数要对字符串数组按照字母进行排序。对列表排序有几个不同的方法, 它们的性能取决于要排序的列表特性。使用委托可以把一个排序算法函数委托传递给排序函数, 指定要使用的方法。

委托有许多用途, 但如前所述, 它们的大多数常见用途主要与事件处理有关, 具体内容详见第13章。

## 6.7 小结

本章相当全面地介绍了 C# 代码中函数的用法。函数提供的其他许多特性(特别是委托)比较抽象, 我们将在第8章的面向对象编程中讨论它们。

如何使用函数的知识是将来要完成的所有编程工作的核心。后面的章节, 特别是学习 OOP(从第8章开始)的部分, 将介绍函数的正式结构, 以及如何把它们应用于类。从现在开始, 把代码放在可重用块中将成为 C# 编程中最有用的部分。

## 6.8 练习

(1) 下面两个函数都存在错误, 请指出这些错误。

```
static bool Write()
{
    Console.WriteLine("Text output from function.");
}

static void myFunction(string label, params int[] args, bool showLabel)
{
    if (showLabel)
        Console.WriteLine(label);
    foreach (int i in args)
```

```
        Console.WriteLine("{0}", i);
    }
```

- (2) 编写一个应用程序，该程序使用两个命令行参数，分别把值放在一个字符串和一个整型变量中，然后显示这些值。
- (3) 创建一个委托，在请求用户输入时，使用它模拟 Console.ReadLine()函数。
- (4) 修改下面的结构，使之包含一个返回订单总价格的函数。

```
struct order
{
    public string itemName;
    public int    unitCount;
    public double unitCost;
}
```

- (5) 在 order 结构中添加另一个函数，该结构返回一个格式化的字符串(一行文本，以合适的值替换用尖括号括起来的斜体条目)。

```
Order Information: <unit count> <item name> items at $<unit cost> each,
total cost $<total cost>
```

附录 A 给出了练习答案。

6.9 本章要点

主 题	重 要 概 念
定义函数	函数用函数名、0 个或多个参数及返回类型来定义。函数的名称和参数统称为函数的签名。可以定义名称相同、但签名不同的多个函数——这称为函数的重载。也可以在结构类型中定义函数
返回值和参数	函数的返回类型可以是任意类型，如果函数没有返回值，其返回类型就是 void。参数也可以是任意类型，由一个用逗号分隔的类型和名称对组成。调用函数时，所指定的参数的类型和顺序必须匹配函数的定义。个数不定的特定类型的参数可以通过参数数组来指定。参数可以指定为 ref 或 out，以便给调用者返回值
变量作用域	变量根据定义它们的代码块来界定其使用范围。代码块包括方法和其他结构，例如循环体。可以在不同的作用域中定义多个不同的同名变量
命令行参数	在执行应用程序时，控制台应用程序中的 Main()函数可以接收传送给应用程序的命令行参数。这些参数用空格隔开，但较长的参数可以放在引号中传送
委托	除了直接调用函数之外，还可以通过委托调用它们。委托是用返回类型和参数列表定义的变量。给定的委托类型可以匹配返回类型和参数与委托定义相同的方法

# 第 7 章

## 调试和错误处理

本章内容:

- IDE 中的调试方法
- C#中的错误处理技术

本书到目前为止介绍了在 C#中进行简单编程的所有基础知识。在讨论本书后面章节的面向对象编程之前,先看看 C#代码中的调试和错误处理问题。

代码中有时难免存在错误。无论程序员多么优秀,程序总是会出现一些问题,出色的程序员会找出其中一部分错误,并更正它们。当然,一些问题比较小,不会影响应用程序的执行,例如,按钮上的拼写错误等,但一些错误可能比较严重,会导致应用程序完全失败(通常称为致命错误),致命错误包括妨碍代码编译的简单错误(语法错误),或者只在运行期间发生的更严重的错误。一些错误可能会更微妙。也许应用程序不能给数据库添加一个记录,因为遗漏了一个请求的字段,或者在其他有限制的环境中把错误的数据添加到记录中。应用程序的逻辑在某些方面有瑕疵时,就会产生这样的错误,此类错误称为语义错误(或逻辑错误)。

当应用程序的用户抱怨说程序不能正常工作时,就出现了比较难以处理的错误。此时需要跟踪代码,确定发生了什么问题,并修改代码,使之按照希望的那样工作。在此类情况下,VS 和 VCE 的调试功能就可以大显身手了。本章的第一部分就介绍一些调试技巧,并把它们应用到一些常见问题上。

接着,讨论 C#中的错误处理技术。利用它们,可以对可能发生错误的地方采取预防措施,并编写弹性代码来处理可能会致命的错误。这些是 C#语言的一部分,而不是调试功能,但 IDE 也提供了一些工具来帮助我们处理错误。

### 7.1 VS 和 VCE 中的调试

前面提到,可以采用两种方式执行应用程序:调试模式或非调试模式。在 VS 或 VCE 中执行应用程序时,它默认在调试模式下执行。例如,按下 F5 键或单击工具栏中的绿色 Play 按钮时,

就是在调试模式下执行应用程序。要在非调试模式下执行应用程序，应选择 **Debug | Start Without Debugging**，或者按下 **Ctrl+F5** 键。

VS 和 VCE 都允许在两种配置下创建应用程序：调试(默认)和发布(实际上，还可以定义其他配置，但这是一种高级技术，本书不涉及)。使用标准工具栏中的 **Solution Configurations** 下拉框可以在这两种配置之间切换。



在 VCE 中，默认情况下不激活这个下拉列表。为了阅读本章，应启用它，方法是选择 **Tools | Options**，在 **Options** 对话框中选择 **Show All Settings**，再选择 **Projects and Solutions** 类别中的 **General** 子类别，启用 **Show Advanced Build Configurations** 选项。

在调试配置下生成应用程序，在调试模式下运行程序时，并不仅仅是运行编写好的代码。调试程序包含了应用程序的符号信息，所以 IDE 知道执行每行代码时发生了什么。符号信息意味着跟踪(例如)未编译代码中使用的变量名，这样，它们就可以匹配已编译的机器码应用程序中现有的值，而机器码程序不包含人们易于读取的信息。此类信息包含在 **.pdb** 文件中，这些文件位于计算机的 **Debug** 目录下。它们可以执行许多有用的操作，包括：

- 向 IDE 输出调试信息
- 在执行应用程序期间查看和编辑变量的值
- 暂停程序和重启程序
- 在代码的某个位置自动暂停程序的执行
- 一次执行程序中的一行代码
- 在应用程序的执行期间监视变量内容的变化
- 在运行期间修改变量内容
- 测试函数的调用

在发布配置中，优化应用程序代码，但我们不能执行这些操作。但发布版本运行得比较快，完成了应用程序的开发时，一般应给用户提提供发布版本，因为发布版本不需要调试版本所包含的符号信息。

本节介绍调试技巧，以及如何使用它们确定未按预期方式执行的那些代码，并修改它们，这个过程称为调试。按照这些技术的使用方法把它们分为两个部分。一般情况下，可以先中断程序的执行，再进行调试，或者注上标记，以便以后加以分析。在 VS 和 VCE 术语中，应用程序可以处于运行状态，也可以处于中断模式，即暂停正常的执行。下面首先介绍非中断模式(运行期间或正常执行)技术。

### 7.1.1 非中断(正常)模式下的调试

本书常常使用的一个命令是 **Console.WriteLine()** 函数，它可以把文本输出到控制台上。在开发应用程序时，这个函数可以方便地获得操作的额外反馈，例如：

```
Console.WriteLine("MyFunc() Function about to be called.");
MyFunc ("Do something.");
Console.WriteLine("MyFunc() Function execution completed.");
```

这段代码说明了如何获取 **MyFunc()** 函数的额外信息。这么做完全正确，但控制台的输出结果会

比较混乱。在开发其他类型的应用程序时，如 Windows 窗体应用程序，没有用于输出信息的控制台。作为一种替代方法，可以把文本输出到另一个位置上——IDE 中的 Output 窗口。

第2章简要介绍了 Error List 窗口，其他窗口也可以显示在这个位置上。其中一个窗口就是 Output 窗口，在调试时这个窗口非常有用。要显示这个窗口，可以选择 View | Output。在这个窗口中，可以查看与代码的编译和执行相关的信息，包括在编译过程中遇到的错误等，还可以将自定义的诊断信息直接写到窗口中，来使用这个窗口显示自定义信息。该窗口如图 7-1 所示。

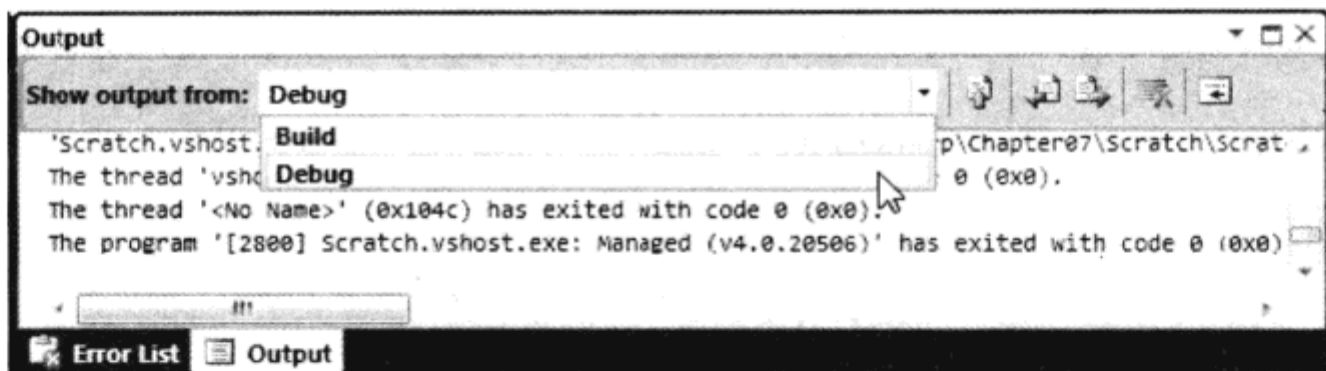


图 7-1



Output 窗口有两种模式 Build 和 Debug，使用其中的下拉菜单可以选择这些模式。Build 和 Debug 模式分别显示编译和运行期间的信息。本节提到“写入 Output 窗口”时，实际上是指“写入 Output 窗口的 Debug 模式视图”。

另外，还可以创建一个日志文件，在运行应用程序时，会把信息添加到该日志文件中。把信息写入日志文件所用的技巧与把文本写到 Output 窗口上所用的技巧相同，但需要理解如何从 C# 应用程序中访问文件系统。我们把这个功能放在后面的章节中，因为目前不必了解文件访问技巧也可以完成很多工作。

### 1. 输出调试信息

在运行期间把文本写入 Output 窗口是非常简单的。只要用需要的调用替代 Console.WriteLine() 调用，就可以把文本写到希望的地方。此时可以使用如下两个命令：

- Debug.WriteLine()
- Trace.WriteLine()

这两个命令函数的用法几乎完全相同，但有一个重要区别。第一个命令仅在调试模式下运行，而第二个命令还可用于发布程序。实际上，Debug.WriteLine() 命令甚至不能编译为可发布的程序，在发布版本中，该命令会消失，这肯定有其优点(首先，编译好的代码文件比较小)。实际上，一个源文件可以创建出两个版本的应用程序。调试版本显示所有的额外诊断信息，而发布版本没有这个开销，也不向用户显示信息，否则会引起用户的反感。

这两个函数的用法与 Console.WriteLine() 是不同的。其唯一的字符串参数用于输出消息，而不需要使用 {X} 语法插入变量值。这意味着必须使用 + 等串联运算符在字符串中插入变量值。它们还可以有第二个字符串参数，用于显示输出文本的类别，这样，如果应用程序的不同地方输出了类似的消息，我们就可以马上确定 Output 窗口中显示的是哪些输出信息。

这些函数的一般输出如下所示:

```
<category>: <message>
```

例如, 下面的语句把 MyFunc 作为可选的类别参数:

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

其结果为:

```
MyFunc: Added 1 to i
```

下面的示例按这种方式输出调试信息。

试一试: 把文本输出到 Output 窗口

- (1) 在 C:\BegVCSharp\Chapter07 目录中创建一个新的控制台应用程序 Ch07Ex01。
- (2) 修改代码, 如下所示:



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace Ch07Ex01
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
            int[] maxValIndices;
            int maxVal = Maxima(testArray, out maxValIndices);
            Console.WriteLine("Maximum value {0} found at element indices:",
                             maxVal);
            foreach (int index in maxValIndices)
            {
                Console.WriteLine(index);
            }
            Console.ReadKey();
        }
        static int Maxima(int[] integers, out int[] indices)
        {
            Debug.WriteLine("Maximum value search started.");
            indices = new int[1];
            int maxVal = integers[0];
            indices[0] = 0;
            int count = 1;
            Debug.WriteLine(string.Format(
                "Maximum value initialized to {0}, at element index 0.", maxVal))
            for (int i = 1; i < integers.Length; i++)
            {
                Debug.WriteLine(string.Format(
```



```

        "Now looking at element at index {0}.", i));
    if (integers[i] > maxVal)
    {
        maxVal = integers[i];
        count = 1;
        indices = new int[1];
        indices[0] = i;
        Debug.WriteLine(string.Format(
            "New maximum found. New value is{0},at element index{1}. " ,
            maxVal, i));
    }
    else
    {
        if (integers[i] == maxVal)
        {
            count++;
            int[] oldIndices = indices;
            indices = new int[count];
            oldIndices.CopyTo(indices, 0);
            indices[count - 1] = i;
            Debug.WriteLine(string.Format(
                "Duplicate maximum found at element index {0}.", i));
        }
    }
}
Trace.WriteLine(string.Format(
    "Maximum value {0} found, with {1} " occurrences.", maxVal, count));
Debug.WriteLine("Maximum value search completed.");
return maxVal;
}
}
}

```

代码段 Ch07Ex01\Program.cs

(3) 在 Debug 模式下执行代码，结果如图 7-2 所示。

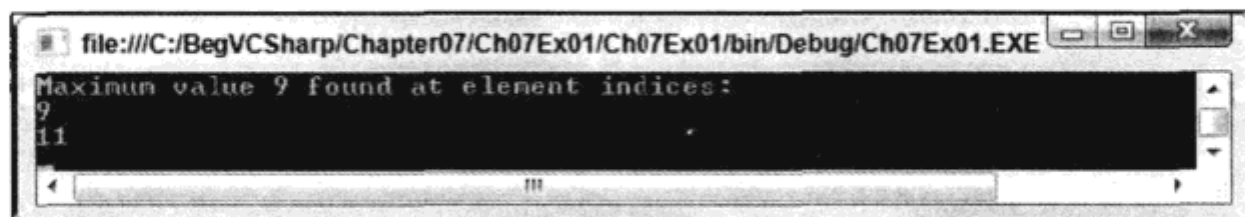


图 7-2

(4) 中断应用程序的执行，查看 Output 窗口中的内容(在 Debug 模式下)，如下所示(有删节)：

```

...
Maximum value search started.
Maximum value initialized to 4, at element index 0.
Now looking at element at index 1.
New maximum found. New value is 7, at element index 1.
Now looking at element at index 2.
Now looking at element at index 3.
Now looking at element at index 4.

```

```

Duplicate maximum found at element index 4.
Now looking at element at index 5.
Now looking at element at index 6.
Duplicate maximum found at element index 6.
Now looking at element at index 7.
New maximum found. New value is 8, at element index 7.
Now looking at element at index 8.
Now looking at element at index 9.
New maximum found. New value is 9, at element index 9.
Now looking at element at index 10.
Now looking at element at index 11.
Duplicate maximum found at element index 11.
Maximum value 9 found, with 2 occurrences.
Maximum value search completed.
The thread 'vshost.RunParkingWindow' (0x110c) has exited with code 0 (0x0).
The thread '<No Name>' (0x688) has exited with code 0 (0x0).
The program '[4568] Ch07Ex01.vshost.exe: Managed' (v4.0.20506) ' has exited with code 0 (0x0).

```

(5) 使用标准工具栏上的下拉菜单，切换到 **Release** 模式，如图 7-3 所示。

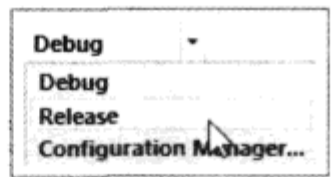


图 7-3

(6) 再次运行程序，这次是在 **Release** 模式下运行，并在执行中止时，再查看一下 **Output** 窗口。结果如下所示(有删节)：

```

...
Maximum value 9 found, with 2 occurrences.
The thread 'Vshost.RunParkingWindow' (0xa78) has exited with code 0 (0x0).
The thread '<No Name>' (0x130c) has exited with code 0 (0x0).
The program '[4348] Ch07Ex01.vshost.exe: Managed (v4.0.20506)' has exited with code 0 (0x0).

```

#### 示例的说明

这个应用程序是第 6 章中一个示例的扩展版本，它使用一个函数计算整数数组中的最大值。这个版本也返回一个索引数组，表示最大值在数组中的位置，以便调用代码处理这些元素。

首先在代码开头使用了一个额外的 **using** 指令：

```
using System.Diagnostics;
```

这简化了本例前面讨论的函数访问，因为它们包含在 **System.Diagnostics** 名称空间中，没有这个 **using** 语句，下面的代码：

```
Debug.WriteLine("Bananas");
```

就需要进一步加以限定，重新编写这行语句，如下所示：

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

**using** 语句使代码更简单，缩短了代码的长度。

**Main()** 中的代码仅初始化一个测试用的整数数组 **testArray**，并声明了另一个整数数组 **maxValIndices**，以存储 **Maxima()** 的索引输出结果(执行计算的函数)，接着调用这个函数。函数返回后，代码就会输出结果。

**Maxima()** 稍复杂一些，但没有使用前面介绍的那么多代码。在数组中进行搜索的方式与第 6 章

的 `MaxVal()` 函数类似，但要用一个记录存储最大值的索引。

特别需要注意用来跟踪索引的函数(而不是输出调试信息的那些代码行)。`Maxima()` 并没有返回一个足以存储源数组中每个索引的数组(需要与源数组有相同的维数)，而是返回一个正好能容纳搜索到的索引的数组。这可以在搜索过程中连续重建不同长度的数组来实现。这是必要的，因为一旦创建好数组，就不能重新设置长度。

开始搜索时，假定源数组(本地称为 `integers`)中的第一个元素就是最大值，且数组中只有一个最大值。因此可以为 `maxVal`(函数的返回值，即搜索到的最大值)和 `indices`(out 参数数组，存储搜索到的最大值的索引)设置值。`MaxVal` 被赋予 `integers` 中第一个元素的值，`indices` 被赋予一个值 0，即数组中第一个元素的索引。在变量 `count` 中存储搜索到的最大值的个数，以跟踪 `indices` 数组。

函数的主体是一个循环，它迭代 `integers` 数组中的各个值，但忽略第一个值，因为它已经处理过了。每个值都与 `maxVal` 的当前值进行比较，如果 `maxVal` 更大，就忽略该值。如果当前处理的值比 `maxVal` 大，就修改 `maxVal` 和 `indices`，以反映这种情况。如果当前处理的值与 `maxVal` 相等，就递增 `count`，用一个新数组替代 `indices`。这个新数组比旧 `indices` 数组多一个元素，包含搜索到的新索引。

最后一个功能的代码如下所示：

```
if (integers[i] == maxVal)
{
    count++;
    int[] oldIndices = indices;
    indices = new int[count];
    oldIndices.CopyTo(indices, 0);
    indices[count - 1] = i;
    Debug.WriteLine(string.Format(
        "Duplicate maximum found at element index {0}.", i));
}
```

这段代码把旧 `indices` 数组备份到 `if` 代码块的 `oldIndices` 局部整型数组中。注意使用 `<array>.CopyTo()` 函数把 `oldIndices` 中的值复制到新的 `indices` 数组中。这个函数的参数是一个目标数组和一个用于复制第一个元素的索引，并把所有的值都粘贴到目标数组中。

在代码中，各个文本部分都使用 `Debug.WriteLine()` 和 `Trace.WriteLine()` 函数来进行输出，这些函数使用 `string.Format()` 函数把变量值嵌套在字符串中，其方式与 `Console.WriteLine()` 相同。这比使用 `+` 串联运算符更加高效。

在 `Debug` 模式下运行应用程序时，其最终结果是一个完整的记录，它记述了在循环中计算出结果所采取的步骤。在 `Release` 模式下，仅能看到计算的最终结果，因为没有调用 `Debug.WriteLine()` 函数。

除了这些 `WriteLine()` 函数外，还需要注意其他一些问题。首先是 `Console.Write()` 的等价函数：

- `Debug.Write()`
- `Trace.Write()`

这两个函数使用的语法与 `WriteLine()` 函数相同(一个或两个参数，即一个消息和可选的类别)，但它们是有区别的，因为它们没有添加行尾字符。

还有下列命令：

- `Debug.WriteLineIf()`
- `Trace.WriteLineIf()`
- `Debug.WriteLine()`

● Trace.WriteLine()

这些函数的参数都与没有 if 的对应函数相同，但增加了一个必选参数，且该参数放在列表参数的最前面。这个参数的值为布尔值(或者计算结果为布尔值的表达式)，只有这个值为 true 时，函数才会输出文本。使用这些函数可以有条件地把文本输出到 Output 窗口中。

例如，只需在某些情况下输出调试信息，所以代码中有许多 Debug.WriteLine()语句，它们都取决于具体的条件。如果没有这个条件，就不显示它们，以防 Output 窗口显示多余的信息。

2. 跟踪点

另一种把信息输出到 Output 窗口中的方法是使用跟踪点。这是 VS 的一个功能，而不是 C#的功能，但其作用与使用 Debug.WriteLine()相同。它实际上是输出调试信息且不修改代码的一种方式。



只能在 VS 中使用跟踪点，不能在 VCE 中使用。如果读者使用的是 VCE，就可以跳过本节。

为了演示跟踪点，可以使用它们替代上一个示例中的调试命令(请参阅本章的下载代码中的 Ch07Ex01TracePoints 文件)。添加跟踪点的过程如下所示：

- (1) 把光标放在要插入跟踪点的代码行上。注意，跟踪点会在执行这行代码之前被处理。
- (2) 右击该行代码，选择 Breakpoint | Insert Tracepoint。
- (3) 在打开的 When Breakpoint Is Hit 对话框中，在 Print a message:文本框中键入要输出的字符串。如果要输出变量值，应把变量名放在花括号中。
- (4) 单击 OK 按钮。在包含跟踪点的代码行左边会出现一个红色的菱形，该行代码也会突出显示为红色。

看一下添加跟踪点的对话框标题和所需要的菜单选项，显然，跟踪点是断点的一种形式(可以暂停应用程序的执行，就像断点一样)。断点一般用于更高级的调试目的，本章稍后将介绍断点。

图 7-4 显示了 Ch07Ex01TracePoints 中第 31 行所需的跟踪点。在删除已有的 Debug.WriteLine()语句后，对代码行编号。

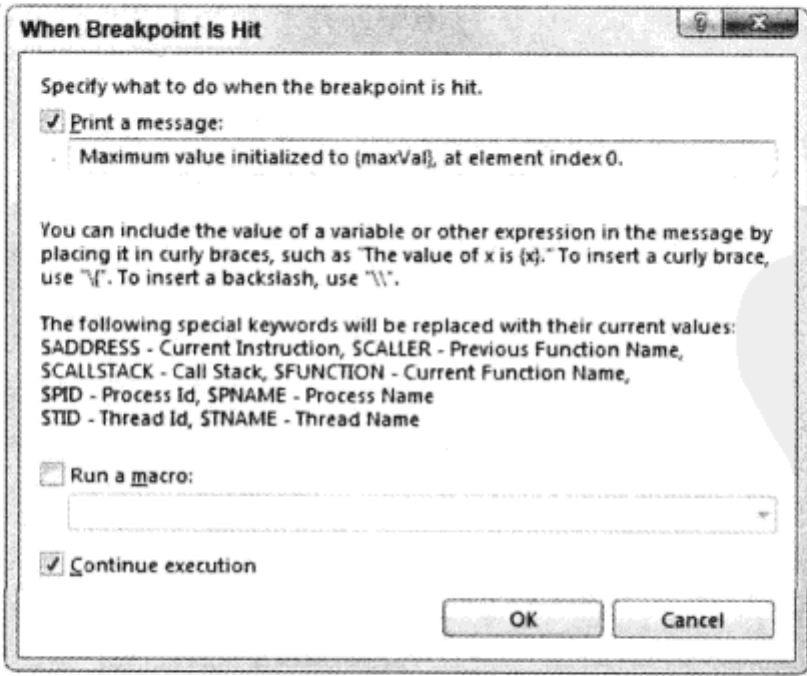


图 7-4



如图 7-4 中的文本所示，跟踪点允许插入与跟踪点的位置和上下文相关的其他有用信息。用户应试着使用这些值，尤其是\$FUNCTION 和\$CALLER，看看可以得到什么额外信息。还可以看出，跟踪点可以执行宏，但这是一个高级功能，这里不予以介绍。

还有一个窗口(只能在 VS 中使用)可用于快速查看应用程序中的跟踪点。要显示这个窗口，可以从 VS 菜单中选择 Debug | Windows | Breakpoints。这是显示断点的通用窗口(如前所述，跟踪点是断点的一种形式)。可以定制显示的内容，从这个窗口的 Columns 下拉框中添加 When Hit 列，显示与跟踪点关系更密切的信息。图 7-5 显示的窗口配置了这个列，还显示了添加到 Ch07Ex01TracePoints 中的所有跟踪点。



图 7-5

在调试模式下执行这个应用程序，会得到与前面完全相同的结果。在代码窗口中右击跟踪点，或者利用 Breakpoints 窗口，就可以删除或临时禁用跟踪点。在 Breakpoints 窗口中，跟踪点左边的复选框确定是否启用跟踪点；禁用的跟踪点未被选中，在代码窗口中显示为菱形框，而不是实心菱形。

### 3. 诊断输出与跟踪点

前面介绍了两种输出相同信息的方法，下面看看它们的优缺点。首先，跟踪点与 Trace 命令并不等价，也就是说，不能使用跟踪点在发布版本中输出信息。这是因为跟踪点并没有包含在应用程序中。跟踪点由 VS 处理，在应用程序的已编译版本中，跟踪点是不存在的。只有应用程序运行在 VS 调试器中时，跟踪点才起作用。

跟踪点的主要缺点也是其优点，即它们存储在 VS 中，因此可以在需要时快速、方便地添加到应用程序中，而且也非常容易删除。如果输出非常复杂的信息字符串，觉得跟踪点非常讨厌，只需单击表示其位置的红色菱形，就可以删除跟踪点。

跟踪点的一个优点是允许方便地添加额外的信息，如上一节提到的 \$FUNCTION。这个信息可以用 Debug 和 Trace 命令来编写，但比较难。总之，输出调试信息的两种方法是：

- **诊断输出：**总是要从应用程序中输出调试结果时使用这种方法，尤其是在要输出的字符串比较复杂，涉及几个变量或许多信息的情况下，使用该方法比较好。另外，如果要在发布模式下获得执行应用程序的调试结果，Trace 命令常常是唯一的选择。
- **跟踪点：**调试应用程序时，希望快速输出重要信息，以便解决语义错误，应使用跟踪点。另一个明显的区别是跟踪点只能在 VS 中使用，而诊断输出可以在 VS 和 VCE 中使用。

#### 7.1.2 中断模式下的调试

调试技术的剩余内容是在中断模式下工作。可以通过几种方式进入这种模式，这些方式都可以暂停程序的执行。

##### 1. 进入中断模式

进入中断模式的最简单方式是在运行应用程序时，单击 IDE 中的 Pause 按钮。这个 Pause 按钮在 Debug 工具栏上，应把该工具栏添加到 VS 默认显示的工具栏中。为此，右击工具栏区域，并选择 Debug，这个工具栏如图 7-6 所示。



图 7-6

在这个工具栏上，前 4 个按钮可以手工控制中断。在图 7-6 上，其中的 3 个按钮显示为灰色，因为在程序没有运行时，它们是不能工作的。还有一个按钮 Start 是可以使用的，这个按钮与标准工具栏上的 Start 按钮相同。在后面的章节需要其他的按钮时，再介绍它们。

运行一个应用程序时，工具栏就如图 7-7 所示。



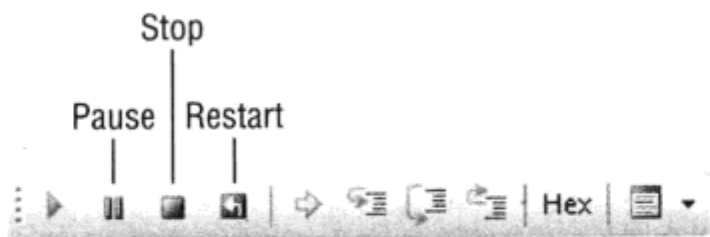


图 7-7

现在，就可以使用之前显示为灰色的 3 个按钮了。它们可以：

- 暂停应用程序的执行，进入中断模式
- 完全停止应用程序的执行(不进入中断模式，而是退出应用程序)
- 重新启动应用程序

暂停应用程序是进入中断模式的最简单方式，但这并不能更好地控制停止程序运行的位置。我们可能会很自然地停止运行应用程序，例如，要求用户输入信息。还可以在长时间的操作或循环过程中进入中断模式，但停止的位置可能相当随机。一般情况下，最好使用断点。

断点

断点是源代码中自动进入中断模式的一个标记，可以在 VS 和 VCE 中使用，但断点在 VS 中更加灵活。它们可以配置为：

- 在遇到断点时，立即进入中断模式
- (只用于 VS)在遇到断点时，如果布尔表达式的值为 true，就进入中断模式
- (只用于 VS)遇到某断点一定的次数后，进入中断模式
- (只用于 VS)在遇到断点时，如果自从上次遇到断点以来变量的值发生了变化，就进入中断模式
- (只用于 VS)把文本输出到 Output 窗口中，或者执行一个宏(参见本章上一节)

注意，上述功能仅能用于调试程序。如果编译发布程序，将会忽略所有断点。

添加断点有几种方法。要添加简单断点，当遇到该断点所在的代码行时，就中断执行，可以单击该代码行左边的灰色区域，右击该代码行，选择 Breakpoint | Insert Breakpoint 菜单项；选择 Debug | Toggle Breakpoint；或者按下 F9 键。

断点在该代码行的旁边显示为一个红色的圆圈，而该行代码也突出显示，如图 7-8 所示。

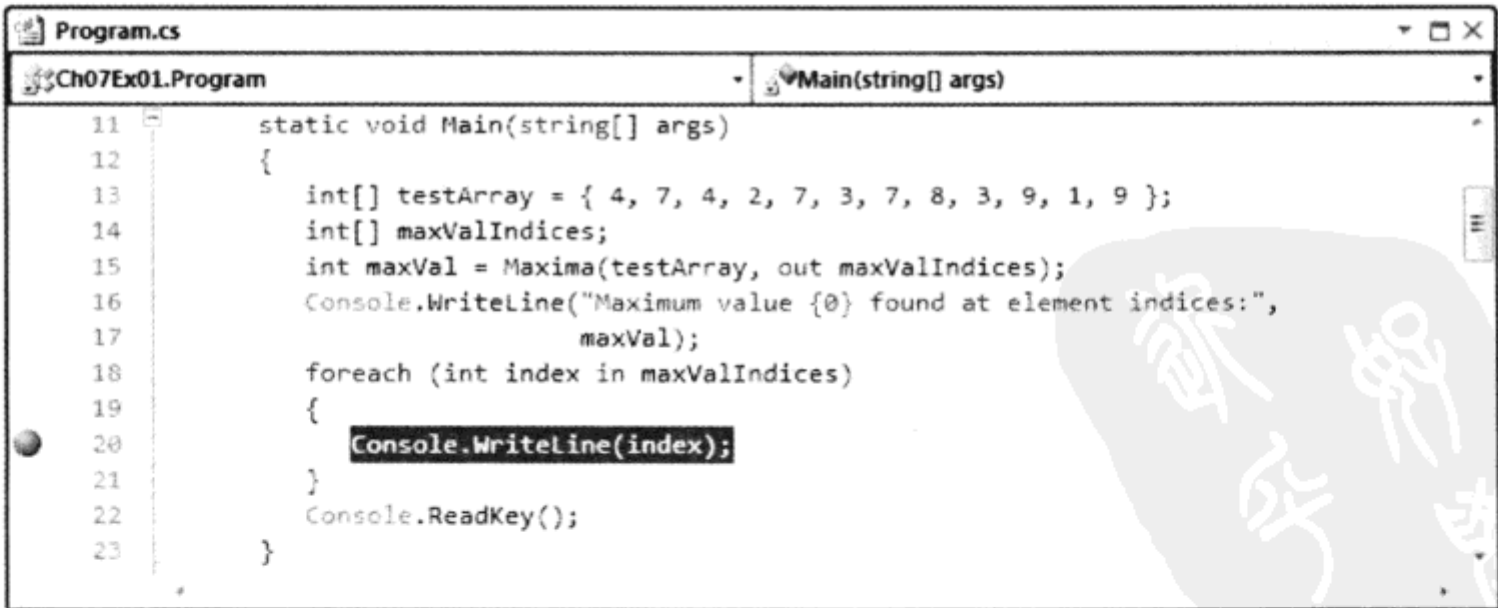


图 7-8

本节的剩余内容仅适用于 VS，不适用于 VCE。如果使用的是 VCE，可以跳到“进入中断模式的其他方式”一节。

在 VS 中，使用 Breakpoints 窗口还可以查看文件中的断点信息(在“跟踪点”一节中介绍过启用该窗口的方法)。在 Breakpoints 窗口中，可以禁用断点(删除描述信息左边的记号；禁用的断点用未填充的红色圆圈来表示)，删除断点，编辑断点的属性。

这个窗口中显示的 Condition 和 Hit Count 列是唯一的两个可用列，它们是非常有用的。右击断点(在代码或这个窗口中)，选择 Condition 或 Hit Count 菜单项，就可以编辑它们。

选择 Condition 按钮，将弹出一个对话框。在该对话框中可以键入任意布尔表达式，该表达式可以包含断点涉及的任何变量。例如，可以配置一个断点，输入表达式 `maxVal > 4`，选择 Is true 选项，则在遇到这个断点，且 `maxVal` 的值大于 4 时，就会触发该断点。还可以检查这个表达式是否有变化，仅当发生变化时，断点才会被触发(例如，如果在遇到断点时，`maxVal` 的值从 2 改为 6，就会触发该断点)。

选择 Hit Count 按钮，将弹出一个对话框。在这个对话框中可以指定在触发前，要遇到该断点多少次。下拉列表提供了如下选项：

- 总是中断
- 在 Hit Count 等于多少次时中断
- 在 Hit Count 是某个数的倍数时中断
- 在 Hit Count 大于等于多少次时中断

所选的选项与在旁边的文本框中输入的值共同确定断点的行为。这个 Hit Count 按钮在比较长的循环中很有用，例如，在执行了前 5000 次循环后需要中断。如果不这么做，中断并再重新启动 5000 次是很痛苦的。



带有附加属性集(例如，条件或遇到断点的次数)的断点，在显示时略有区别。已配置的断点不是显示一个简单的红色圆圈，而是在红色的圆圈中有一个白色的加号。这是很有用的，因为它允许很快辨认出哪个断点总是进入中断模式，哪个断点只在某种情况下才进入中断模式。

### 进入中断模式的其他方式

进入中断模式还有两种方式。一种是在抛出一个未处理的异常时选择进入该模式。这种方式在本章后面讨论到错误处理时论述。另一种方式是生成一个判定语句(assertion)时中断。

判定语句是可以用用户定义的消息中断应用程序的指令。它们常常用于应用程序的开发过程，作为测试程序是否能平滑运行的一种方式。例如，在应用程序的某一处要求给定的变量值小于 10，此时就可以使用一个判定语句，确定它是否为 true，如果不是，就中断程序的执行。当遇到判定语句时，可以选择 Abort，中断应用程序的执行，也可以选择 Retry，进入中断模式，还可以选择 Ignore，让应用程序像往常一样继续执行。

与前面的调试输出函数一样，判定函数也有两个版本：

- `Debug.Assert()`
- `Trace.Assert()`

其调试版本也是仅用于编译调试程序。

这两个函数带3个参数。第一个参数是一个布尔值，其值为 false 会触发判定语句。第二、三个参数是两个字符串，分别把信息写到弹出的对话框和 Output 窗口中。上面的示例需要一个函数调用，如下所示：

```
Debug.Assert(myVar < 10, "myVar is 10 or greater.",
    "Assertion occurred in Main().");
```

判定语句通常在应用程序的早期使用比较有效。可以分发应用程序的一个发布程序，其中包含 Trace.Assert() 函数，以列出各种信息。如果触发了判定语句，用户就会收到通知，把这些消息传递给开发人员。这样，即使开发人员不知道错误是如何发生的，也可以改正这个错误。

例如，在第一个字符串中提供错误的简短描述，在第二个字符串中提供下一步该如何操作的指示：

```
Trace.Assert(myVar < 10, "Variable out of bounds.",
    "Please contact vendor with the error code KCW001.");
```

如果触发了这个判定语句，用户就会看到如图 7-9 所示的对话框。

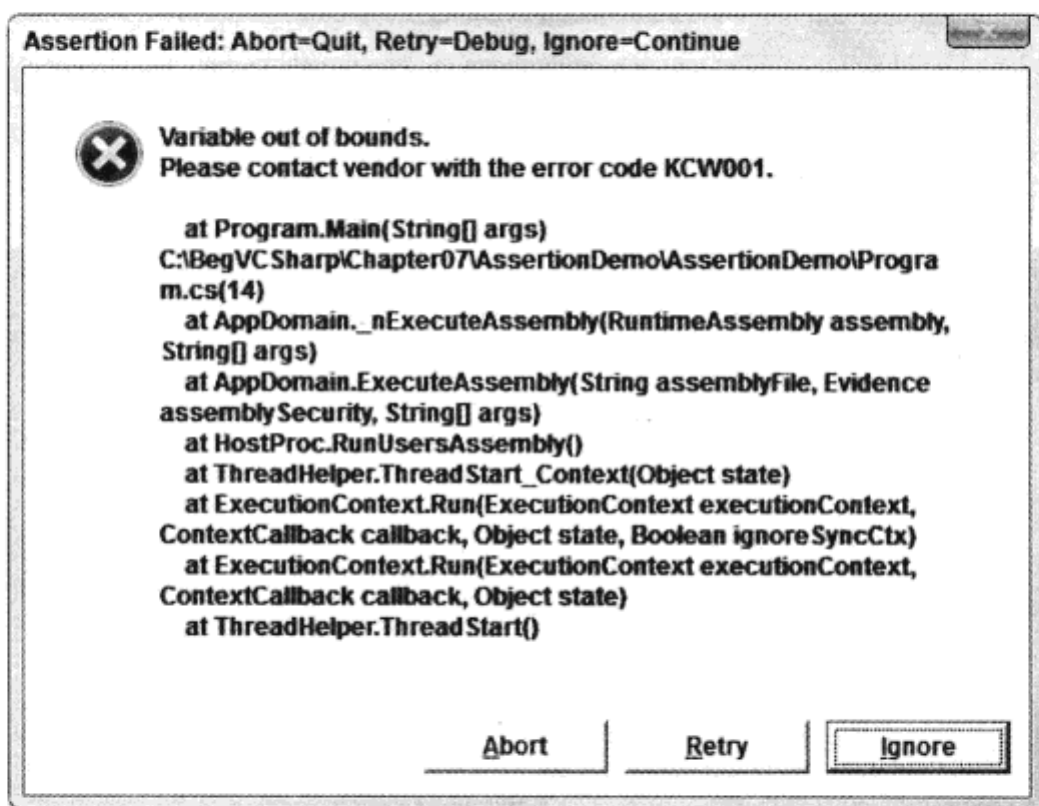


图 7-9

诚然，这并不是最友好的对话框，因为它包含了许多令人迷惑的信息，但如果用户给开发人员发送了错误的屏幕图，开发人员就可以很快找出问题所在。

下一个要论述的主题是应用程序中断，以及进入中断模式后，我们可以做什么。一般情况下，进入中断模式的目的是找出代码中的错误(或确保程序工作正常)。一旦进入中断模式，就可以使用各种技巧分析代码，分析应用程序在暂停处的确切状态。

## 2. 监视变量的内容

监视变量的内容是 VS 和 VCE 帮助我们使工作变得简单的一个方面。查看变量值的最简单方式是在中断模式下，使鼠标指向源代码中的变量名，此时就会出现一个黄色的工具提示，显示该变量

的信息，其中包括该变量的当前值。

还可以高亮显示整个表达式，以相同的方式得到该表达式的结果。对于比较复杂的值，例如，数组，甚至可以扩展工具提示中的值，查看各个数组元素项。

注意，在运行应用程序时，IDE 中各个窗口的布局发生了变化，在默认情况下，运行期间会发生如下变化(变化的情况会根据具体的安装略有区别)：

- Properties 窗口消失，其他一些窗口也会消失，包括 Solution Explorer 窗口
- Error List 窗口会被屏幕底部的两个新窗口代替
- 新窗口中会出现几个新的选项卡

新的屏幕布局如图 7-10 所示。这可能与读者的显示情况不完全相同，一些选项卡和窗口可能不完全匹配。但是，这些窗口的功能(后面将讨论)是相同的，这个显示完全可以通过 View 和 Debug | Windows 菜单来定制(在中断模式下)，也可以在屏幕上拖动窗口，重新设定它们的位置。

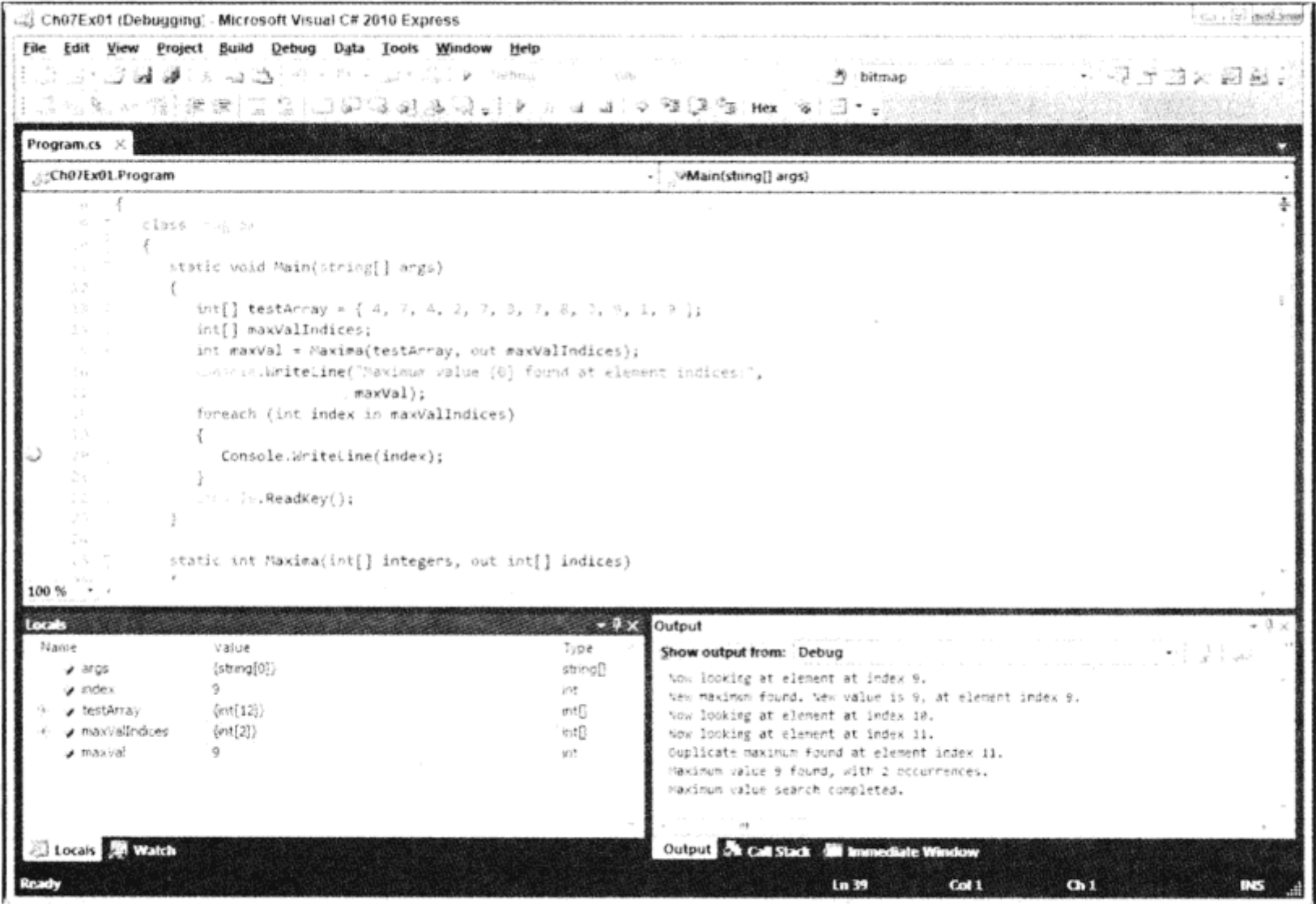


图 7-10

底部左角边的新窗口在调试时非常有用，它允许在中断模式下，在应用程序的变量值上保留标签，它包含 3 个选项卡，如下所示(在 VS 和 VCE 中有所不同)：

- Autos(只在 VS 中有)——当前和前面的语句使用的变量(Ctrl+D, A)
- Locals——作用域内的所有变量(Ctrl+D, L)
- Watch N——可定制的变量和表达式显示(其中 N 从 1~4，在 Debug Windows Watch 上)

这些选项卡的工作方式或多或少有些类似，并根据它们的特定功能添加了各种附加特性。一般情况下，每个选项卡都包含一个变量列表，其中包括变量的名称、值和类型等信息。更复杂的变量(如

数组)可以使用变量名左边的+和-(展开/折叠)符号进一步查看,它们的内容可以以树状视图的方式显示。例如,在前面的示例中,在代码中放置了一个断点,得到的 Locals 选项卡如图 7-11 所示,其中显示了数组变量 maxValIndices 的展开视图。

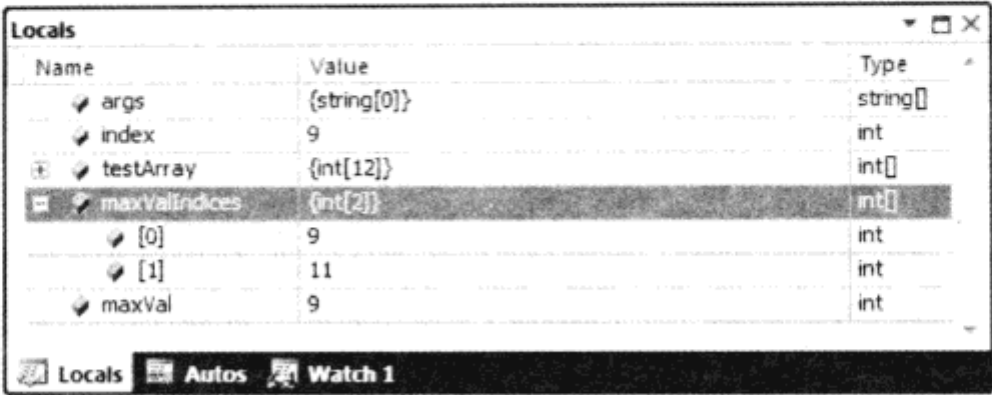


图 7-11

在这个视图中,还可以编辑变量的内容。它有效地绕过了前面代码中的其他变量赋值。为此,只需在 Value 列中为要编辑的变量输入一个新值即可。也可以把这种技巧用于其他情况,例如,需要修改代码才能编辑变量值的情况。

可以通过 Watch 窗口(或 VS 中的 Watch 窗口,至多可以显示 4 个)监视特定变量或涉及特定变量的表达式。要使用这个窗口,只需在 Name 列中键入变量名或表达式,就可以查看它们的结果,注意,并不是应用程序中的所有变量在任何时候都在作用域内,并在 Watch 窗口中对变量做出标记。例如,图 7-12 显示了一个 Watch 窗口,其中包含几个示例变量和表达式,在遇到 Maxima()函数末尾前面的一个断点时,会显示这个 Watch 窗口。

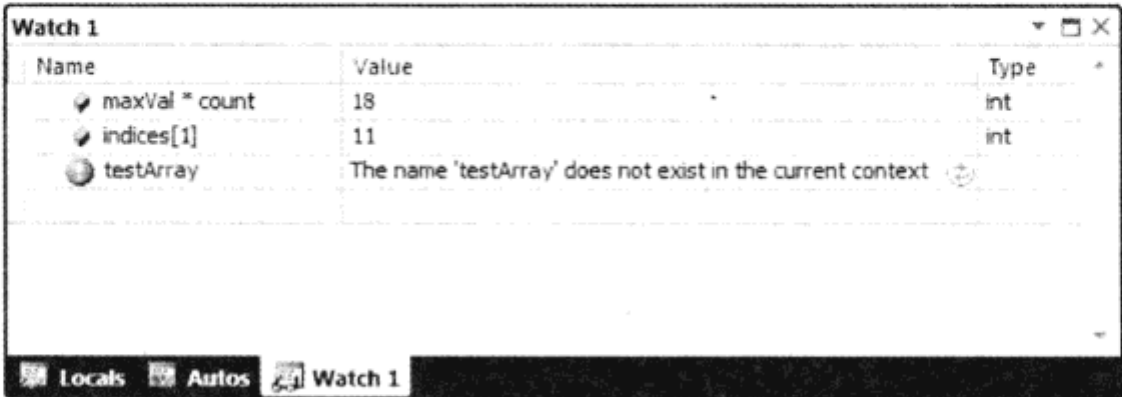


图 7-12

testArray 数组对于 Main()来说是局部数组,所以在该图中没有值,而是显示了一个信息,告诉我们这个变量不在作用域内。



要在 Watch 窗口中添加变量,还可以把变量从源代码拖动到该窗口中。

在这个窗口中可以访问变量的各种显示结果,一个优点是它们可以显示变量在断点之间的变化情况。新值显示为红色而不是黑色,所以很容易看出哪个值发生了变化。

如前所述,要在 VS 中添加更多的 Watch 窗口,可以在中断模式下,使用 Debug | Windows | Watch | Watch N 菜单选项打开或关闭 Watch 的 4 个窗口。每个窗口都可以包含变量和表达式的一组观察结果,所以可以把相关的变量组合在一起,以便于访问。



除了这些 Watch 窗口外，VS 还有一个 QuickWatch 窗口，它能快速提供源代码中某个变量的详细信息。要使用这个窗口，可以右击要查看的变量，选择 QuickWatch 菜单选项。但在大多数情况下，使用标准的 Watch 窗口就足够了。

Watch 窗口可以在应用程序的各个执行过程之间保留下来。如果中断应用程序，再重新运行，就不必再次添加 Watch 窗口了，IDE 会记住上次使用的 Watch 窗口。

3. 单步执行代码

前面介绍了如何在中断模式下查看应用程序的运行情况，下面论述如何在中断模式下使用 IDE 单步执行代码，查看代码的执行结果，人们的思维速度不会比计算机运行得更快，所以这是一个极有价值的技巧。

进入中断模式后，在代码视图的左边，正在执行的代码旁边会出现一个光标(如果使用断点进入中断模式，该光标最初应显示在断点的红色圆圈中)，如图 7-13 所示。

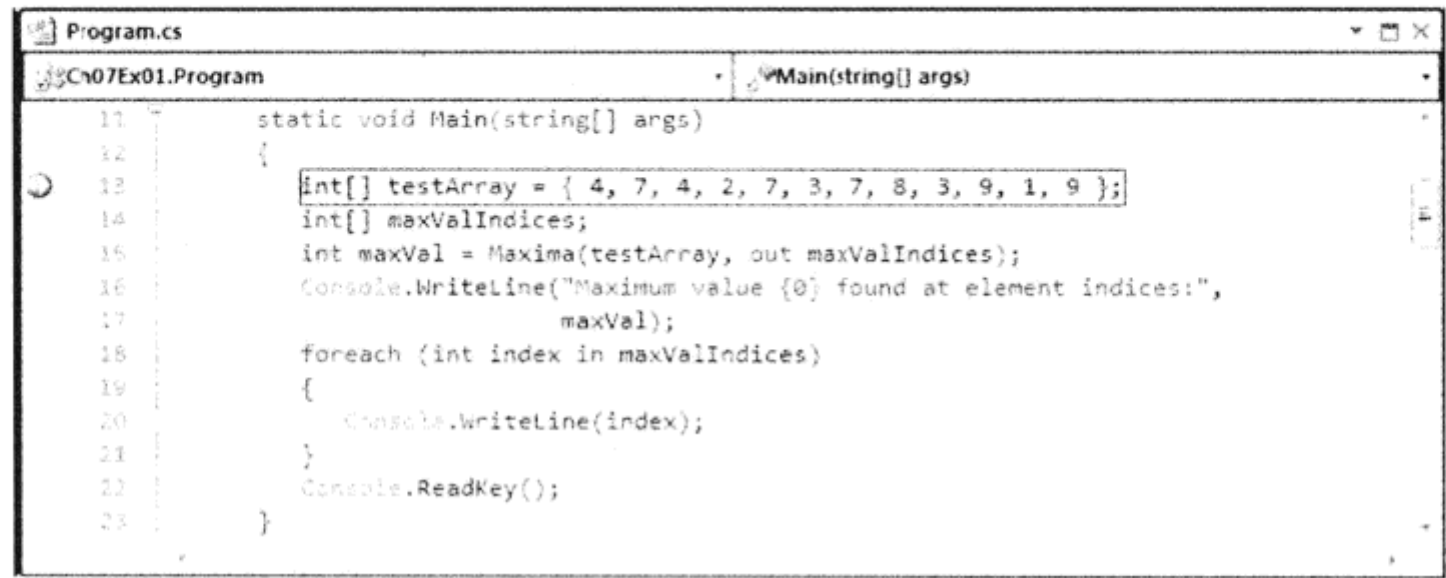


图 7-13

这显示了在进入中断模式时程序执行到的位置。在这个位置上，可以选择逐行执行。为此，使用前面看到的其他一些 Debug 工具栏按钮，如图 7-14 所示。

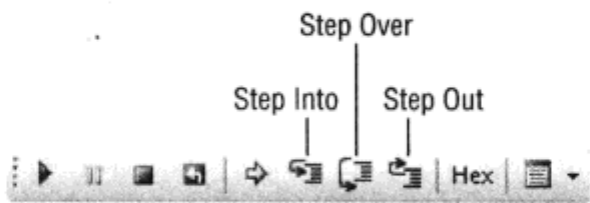


图 7-14

第 6、7、8 个图标控制了中断模式下的程序流。它们依次是：

- Step Into——执行并移动到下一个要执行的语句上
- Step Over——同上，但不进入嵌套的代码块，包括函数
- Step Out——执行到代码块的末尾，在执行完该语句块后，重新进入中断模式

如果要查看应用程序执行的每个操作，可以使用 Step Into 按顺序执行指令，这包括在函数中执行，例如，上面示例中的 Maxima()。当光标到达第 15 行，调用 Maxima()时，单击这个图标，会使光标移动到 Maxima()函数内部的第一行代码上。而如果光标移到第 15 行时单击 Step Over，就会使光标移动到第 16 行，不必进入 Maxima()中的代码(但仍执行这段代码)。如果单步执行到不感兴趣的



函数，可以单击 Step Out，返回到调用该函数的代码。在单步执行代码时，变量的值可能会发生变化。注意观察上一节讨论的 Watch 窗口，可以看到变量值的变化情况。

在存在语义错误的代码中，这个技巧也许是最有效的。可以单步执行代码，当执行到有错误的代码时，错误会像正常运行程序那样发生。在这个过程中，可以监视数据，看看什么地方出错。本章后面将使用这个技巧查看示例应用程序的执行情况。

#### 4. Immediate 和 Command 窗口

Command(只有 VS 中有)和 Immediate 窗口(选择 Debug | Windows 菜单)可以在运行应用程序的过程中执行命令。通过 Command 窗口可以手动执行 VS 操作(例如，菜单和工具栏操作)，Immediate 窗口可以执行源代码，计算表达式，还可以执行其他代码。

VS 中的这些窗口在内部是链接在一起的(实际上，VS 的早期版本把它们当作同一个窗口)。甚至可以在它们之间切换：输入命令 immed，可以从 Command 窗口切换到 Immediate 窗口；输入 >cmd 可以从 Immediate 窗口切换到 Command 窗口。

下面详细讨论 Immediate 窗口，因为 Command 窗口仅适用于复杂的操作，只能在 VS 中使用，而 Immediate 窗口可以在 VS 和 VCE 中使用。Immediate 窗口最简单的用法是计算表达式，有点像 Watch 窗口中的一次性使用。为此，只需键入一个表达式，并按回车键即可。接着就会显示请求的信息，如图 7-15 所示。



图 7-15

也可以在这里修改变量的内容，如图 7-16 所示。

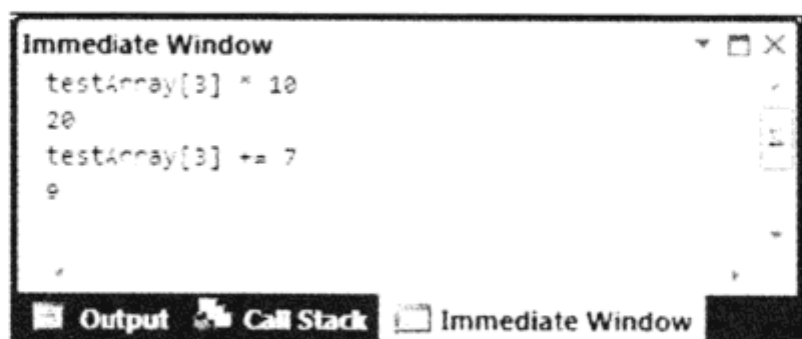


图 7-16

在大多数情况下，使用前面介绍的变量监视窗口更容易得到相同的效果，但这个技巧对于常常发生变化的变量值仍很方便，也适合于测试以后不感兴趣的表达式。

#### 5. Call Stack 窗口

这是最后一个要讨论的窗口，它描述了程序是如何执行到当前位置的。简言之，该窗口显示了当前函数、调用它的函数以及调用函数的函数(即一个嵌套的函数调用列表)。调用的位置也被记录下来。

在前面的示例中，在执行到 `Maxima()` 时进入中断模式，或者使用代码单步执行功能移动到这个函数的内部，得到如图 7-17 所示的信息。

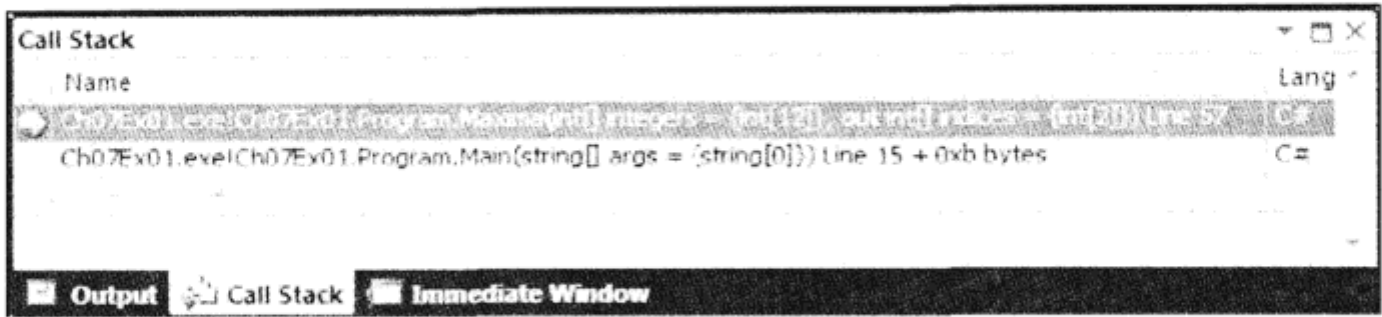


图 7-17

如果双击某一项，就会移动到相应的位置，跟踪代码执行到当前位置的过程。在第一次检测错误时，这个窗口非常有用，因为它们可以查看临近错误发生时的情况。对于常用函数中出现的错误，则有助于找到错误的源头。

有时 Call Stack 窗口会显示一些非常杂乱的信息，例如，有时因为以错误的方式使用了外部函数，错误在应用程序的外部发生，就会出现这种情况。此时，这个窗口中会列出一个很长的列表，其中只有一两个选项是我们熟悉的。如果需要，可以右击该窗口，选择 `Show External Code`，查看外部引用。

## 7.2 错误处理

本章的第一部分讨论如何在应用程序的开发过程中查找和改正错误，使这些错误不会在发布的代码中出现。但有时，我们知道可能会有错误发生，但不能100%地肯定它们不会发生。此时，最好能预料到错误的发生，编写足够强壮的代码以处理这些错误，而不必中断程序的执行。

错误处理就是用于这个目的，下面还将介绍异常和处理它们的方式。异常是在运行期间代码中产生的错误，或者由代码调用的函数产生的错误。这里的“错误”定义要比以前更含糊，因为异常可能是在函数等结构中手工产生。例如，如果函数的一个字符串参数不是以 `a` 开头，就产生一个异常。这并不是严格意义上的函数外部错误，但调用该函数的代码把它看作函数外部错误。

您已在本书前面已经遇到几次异常了。最简单的示例是试图定位一个超出范围的数组元素，例如：

```
int[] myArray = {1, 2, 3, 4};
int myElem = myArray[4];
```

这会产生如下异常信息，并中断应用程序的执行：

```
Index was outside the bounds of the array.
```

前面介绍了异常辅助信息窗口的一些示例。该窗口中的一行把它与出错的代码连接起来，还包含.NET 帮助文件中相关主题的连接和一个 `View Detail` 链接，利用该链接可以找到所发生异常的更多信息。

异常在命名空间中定义，大多数异常的名称清晰地说明了它们的用途。在这个示例中，产生的异常叫做 `System.IndexOutOfRangeException`，说明我们提供的 `myArray` 数组索引不在允许使用的索引范围内。在异常未处理时，这个信息才会显示出来，应用程序也才会中断执行。下一节将讨论如何处理异常。

### 7.2.1 try...catch...finally

C#语言包含结构化异常处理(Structured Exception Handling, SEH)的语法。用3个关键字可以标记出能处理异常的代码和指令，如果发生异常，就使用这些指令处理异常。用于这个目的的3个关键字是 `try`、`catch` 和 `finally`。它们都有一个关联的代码块，必须在连续的代码行中使用。其基本结构如下：

```
try
{
    ...
}
catch (<exceptionType> e)
{
    ...
}
finally
{
    ...
}
```

也可以只有 `try` 块和 `finally` 块，而没有 `catch` 块，或者有一个 `try` 块和好几个 `catch` 块。如果有一个或多个 `catch` 块，`finally` 块就是可选的，否则就是必需的。这些代码块的用法如下：

- **try**——包含抛出异常的代码(在谈到异常时，其中的“抛出”在C#语言中也可以是“生成”或“导致”)。
- **catch**——包含抛出异常时要执行的代码。`catch` 块可以使用 `<exceptionType>`，设置为只响应特定的异常类型(如 `System.IndexOutOfRangeException`)，以便提供多个 `catch` 块。还可以完全省略这个参数，让一般的 `catch` 块响应所有异常。
- **finally**——包含总是会执行的代码，如果没有产生异常，则在 `try` 块之后执行，如果处理了异常，就在 `catch` 块后执行，或者在未处理的异常上移到调用堆栈之前执行。“上移到调用堆栈”表示，SEH 允许嵌套 `try...catch...finally` 块，可以直接嵌套，也可以在 `try` 块包含的函数调用中嵌套。例如，如果在被调用的函数中没有 `catch` 块能处理某个异常，就由调用代码中的 `catch` 块处理。如果始终没有匹配的 `catch` 块，就终止应用程序。`finally` 块在此之前处理，是因为存在这个块，否则也可以在 `try...catch...finally` 结构的外部放置代码。这个嵌套功能将在后面的“异常处理的注意事项”一节中进一步讨论，所以如果对这个功能感到迷惑，请不必担心。

在 `try` 块的代码中出现异常后，发生的事件依次是：

- `try` 块在发生异常的地方中断程序的执行。
- 如果有 `catch` 块，就检查该块是否匹配已抛出的异常类型。如果没有 `catch` 块，就执行 `finally` 块(如果没有 `catch` 块，就一定要有 `finally` 块)。
- 如果有 `catch` 块，但它与已发生的异常类型不匹配，就检查是否有其他 `catch` 块。
- 如果有 `catch` 块匹配已发生的异常类型，就执行它包含的代码，再执行 `finally` 块(如果有)。

- 如果 catch 块都不匹配已发生的异常类型，就执行 finally 块(如果有)。

下面用一个“试一试”示例来说明异常的处理。这个示例以几种方式抛出和处理异常，以便读者了解其工作情况。

### 试一试：异常处理

(1) 在 C:\BegVCSharp\Chapter07 目录中创建一个新控制台应用程序 Ch07Ex02。

(2) 修改代码，如下所示(这里显示的行号注释有助于将代码与后面的讨论相匹配，在本章的可下载代码中复制了它们，以便使用)：



```
class Program
{
    static string[] eTypes = {"none", "simple", "index", "nested index"};

    static void Main(string[] args)
    {
        foreach (string eType in eTypes)
        {
            try
            {
                Console.WriteLine("Main() try block reached."); // Line 23
                Console.WriteLine("ThrowException(\"{0}\") called.", eType); // Line 24
                ThrowException(eType);
                Console.WriteLine("Main() try block continues."); // Line 26
            }
            catch (System.IndexOutOfRangeException e) // Line 28
            {
                Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                    + " block reached. Message:\n\"{0}\",",
                    e.Message);
            }
            catch // Line 34
            {
                Console.WriteLine("Main() general catch block reached.");
            }
            finally
            {
                Console.WriteLine("Main() finally block reached.");
            }
            Console.WriteLine();
        }
        Console.ReadKey();
    }

    static void ThrowException(string exceptionType)
    {
        // Line 49
        Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType);
        switch (exceptionType)
        {
            case "none" :
                Console.WriteLine("Not throwing an exception.");
        }
    }
}
```

```

        break; // Line 54
    case "simple" :
        Console.WriteLine("Throwing System.Exception.");
        throw (new System.Exception()); // Line 57
        break;
    case "index" :
        Console.WriteLine("Throwing System.IndexOutOfRangeException.");
        eTypes[4] = "error"; // Line 60
        break;
    case "nested index" :
        try // Line 63
        {
            Console.WriteLine("ThrowException(\"nested index\") " +
                               "try block reached.");
            Console.WriteLine("ThrowException(\"index\") called.");
            ThrowException("index"); // Line 68
        }
        catch // Line 70
        {
            Console.WriteLine("ThrowException(\"nested index\") general"
                               + " catch block reached.");
        }
        finally
        {
            Console.WriteLine("ThrowException(\"nested index\") finally"
                               + " block reached.");
        }
        break;
    }
}
}

```

代码段 Ch07Ex02\Program.cs

(3) 运行应用程序，结果如图 7-18 所示。

```

file:///C:/BegVCSharp/Chapter07/Ch07Ex02/Ch07Ex02/bin/Debug/Ch07Ex02.EXE
Main() try block reached.
ThrowException("none") called.
ThrowException("none") reached.
Not throwing an exception.
Main() try block continues.
Main() finally block reached.

Main() try block reached.
ThrowException("simple") called.
ThrowException("simple") reached.
Throwing System.Exception.
Main() general catch block reached.
Main() finally block reached.

Main() try block reached.
ThrowException("index") called.
ThrowException("index") reached.
Throwing System.IndexOutOfRangeException.
Main() System.IndexOutOfRangeException catch block reached. Message:
"Index was outside the bounds of the array."
Main() finally block reached.

Main() try block reached.
ThrowException("nested index") called.
ThrowException("nested index") reached.
ThrowException("nested index") try block reached.
ThrowException("index") called.
ThrowException("index") reached.
Throwing System.IndexOutOfRangeException.
ThrowException("nested index") general catch block reached.
ThrowException("nested index") finally block reached.
Main() try block continues.
Main() finally block reached.

```

图 7-18

### 示例的说明

这个应用程序在 Main() 中有一个 try 块，它调用函数 ThrowException()。这个函数会根据调用时使用的参数抛出异常：

- ThrowException("none")——不抛出异常。
- ThrowException("simple")——生成一般异常。
- ThrowException("index")——生成 System.IndexOutOfRangeException 异常。
- ThrowException("nested index")——包含它自己的 try 块，其中的代码调用 ThrowException("index")，生成一个 System.IndexOutOfRangeException 异常。

其中的每个 string 参数都存储在全局数组 eTypes 中，在 Main() 函数中迭代，用每个可能的参数调用 ThrowException()。在迭代过程中，会把各种信息写到控制台上，说明发生了什么情况。这段代码可以使用本章前面介绍的代码单步执行技巧。在执行代码的过程中，一次执行一行代码可以确切地了解代码的执行进度。

在代码的第 23 行添加一个新断点(用默认的属性)，该行代码如下：

```
Console.WriteLine("Main() try block reached.");
```



这里使用了行号，因为它们显示在这段代码的下载版本中。如果关闭了行号，可以在 Text Editor | C# | General 选项区域中，选择 Tools | Options 菜单选项打开它们。上述代码包含注释，这样读者可以阅读文本，而无需打开文件。

在调试模式下运行应用程序。程序立即进入中断模式，此时光标停在第 23 行上。如果选择变量监视窗口中的 Locals 选项卡，就会看到 eType 当前是 none。使用 Step Into 按钮处理第 23 和 24 行，看看第一行文本是否已经写到控制台上。接着使用 Step Into 按钮单步执行第 25 行的 ThrowException() 函数。

执行到 ThrowException() 函数(第 49 行)后，Locals 窗口会发生变化。eType 和 args 不再能访问(因为它们是 Main() 的局部变量)，我们看到的是 exceptionType 局部参数，它当然是 none。继续单击 Step Into，到达 switch 语句，检查 exceptionType 的值，执行代码，把字符串 Not throwing an exception 写到屏幕上。在执行第 54 行上的 break 语句时，将退出函数，继续处理 Main() 中的第 26 行代码。因为没有抛出异常，所以继续执行 try 块。

接着处理 finally 块。再单击 Step Into 几次，执行完 finally 块和 foreach 的第一次循环。下次执行到第 25 行时，使用另一个参数 simple 调用 ThrowException()。

继续使用 Step Into 单步执行 ThrowException()，最终会执行到第 57 行：

```
throw (new System.Exception());
```

这里使用 C# throw 关键字生成一个异常，需要为这个关键字提供新初始化的异常作为其参数，抛出一个异常，这里使用名称空间 System 中的另一个异常 System.Exception。



在这个 case 块中不需要 break 语句，使用 throw 就可以结束该块的执行。



在使用 Step Into 执行这个语句时，将从第 34 行开始执行一般的 catch 块。因为与第 28 行开始的 catch 块都不匹配，所以执行这个一般的 catch 块。单步执行这段代码，然后执行 finally 块，最后返回另一个循环周期，该循环在第 25 行用一个新参数调用 ThrowException()，这次的参数是 index。

这次 ThrowException()在第 60 行生成一个异常：

```
eTypes[4] = "error";
```

eTypes 是一个全局数组，所以可以在这里访问它。但是这里试图访问数组中的第 5 个元素(其索引从 0 开始计数)，这会生成一个 System.IndexOutOfRangeException 异常。

这次 Main()中有一个匹配的 catch 块，单步执行该 catch 块，从第 28 行开始。这个块中调用的 Console.WriteLine()使用 e.Message，输出存储在异常中的信息(可以通过 catch 块的参数访问异常)。之后再次单步执行 finally 块(而不是第二个 catch 块，因为异常已经处理完了)。返回循环，再次调用第 25 行的 ThrowException()。

在执行到 ThrowException()中的 switch 结构时，进入一个新的 try 块，从第 63 行开始。在执行到第 68 行时，将遇到 ThrowException()的一个嵌套调用，这次使用 index 参数。可以使用 Step Over 按钮跳过其中的代码行，因为前面已经单步执行过了。与前面一样，这个调用生成一个 System.IndexOutOfRangeException 异常。但这个异常在 ThrowException()中的嵌套 try...catch...finally 结构中处理。这个结构没有明确匹配这种异常的 catch 块，所以执行一般的 catch 块(从第 70 行开始)。

与前面的异常处理一样，现在单步执行这个 catch 块，以及关联的 finally 块，最后返回到函数调用的末尾。但是它们有一个重大的区别：抛出的异常是由 ThrowException()中的代码处理的。这就是说，异常并没有留给 Main()处理，所以直接进入 finally 块，之后应用程序中断执行。

7.2.2 列出和配置异常

.NET Framework 包含许多异常类型，可以在代码中自由抛出和处理这些类型的异常，甚至可以在代码中抛出异常，让它们在比较复杂的应用程序中被捕获。IDE 提供了一个对话框，可以检查和编辑可用的异常，可以使用 Debug | Exceptions 菜单选项(或按下 Ctrl+D, E)打开该对话框，如图 7-19 所示(如果使用 VCE，则列表中的项会不同，只包含图 7-19 中的第二、三项)。

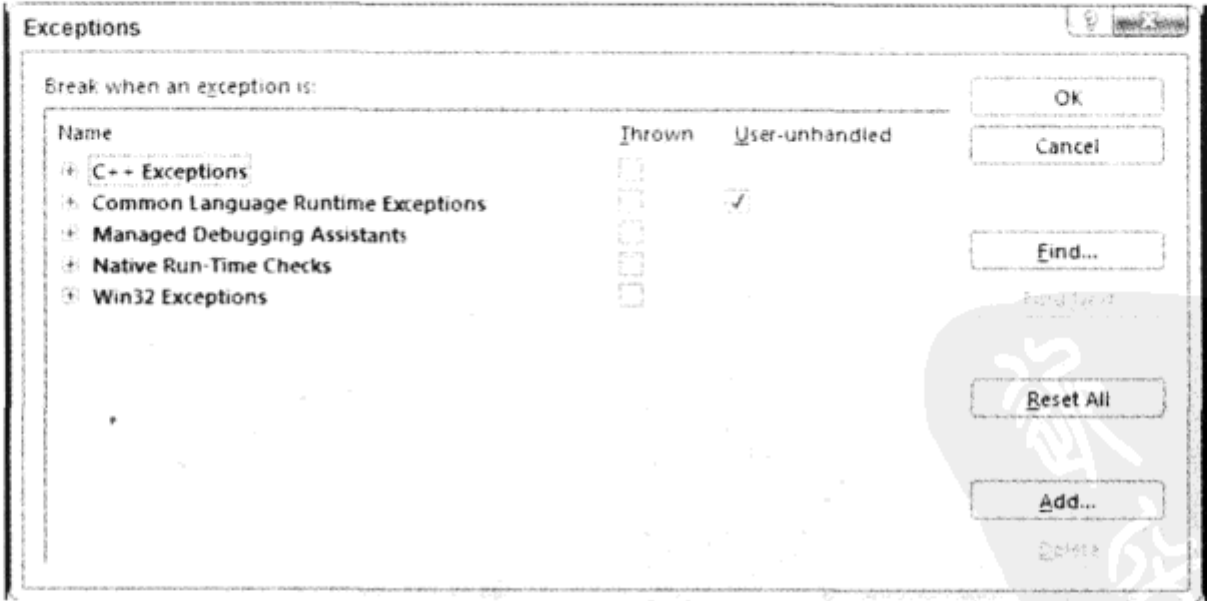


图 7-19

按照类别和.NET 库名称空间列出异常。展开 Common Language Runtime Exceptions 选项，再展开 System 选项，就可以看到 System 名称空间中的异常，这个列表包括上面使用的异常

`System.IndexOutOfRangeException`。

每个异常都可以使用右边的复选框来配置。可以使用第一个选项(break when)Thrown 中断调试器，即使是对于已处理的异常，也是这样。第二个选项可以忽略未处理的异常，这样做会对结果有影响。在大多数情况下，这会进入中断模式，所以只需在异常环境下这么做。

在大多数情况下，使用默认设置就足够了。

### 7.2.3 异常处理的注意事项

注意，必须在更一般的异常捕获之前为比较特殊的异常提供 catch 块。如果 catch 块的顺序错误，应用程序就会编译失败。还要注意可以在 catch 块中抛出异常，方法是使用上一个示例中的方式，或使用下述表达式：

```
throw;
```

这个表达式会再次抛出 catch 块处理过的异常。如果以这种方式抛出异常，该异常就不会由当前的 try...catch...finally 块处理，而是由上一级的代码处理(但嵌套结构中的 finally 块仍会执行)。

例如，如果修改 `ThrowException()` 中的 try...catch...finally 块，如下所示：

```
try
{
    Console.WriteLine("ThrowException(\"nested index\") " +
        "try block reached.");
    Console.WriteLine("ThrowException(\"index\") called.");
    ThrowException("index");
}
catch
{
    Console.WriteLine("ThrowException(\"nested index\") general"
        + " catch block reached.");
    throw;
}
finally
{
    Console.WriteLine("ThrowException(\"nested index\") finally"
        + " block reached.");
}
```

则首先执行其中的 finally 块，再执行 Main() 中匹配的 catch 块，得到的控制台输出如图 7-20 所示。

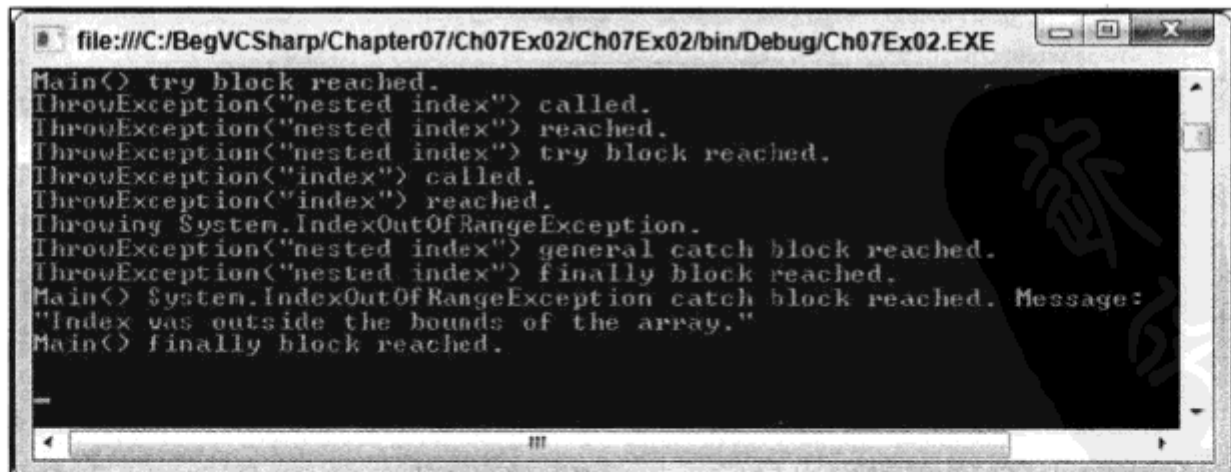


图 7-20

在这个屏幕图中，Main()函数输出了额外的几行，因为这个函数捕获了 System.IndexOutOfRangeException 异常。

7.3 小结

本章主要论述了调试应用程序所使用的技巧。有许多这方面的技巧，其中的大部分可用于各类项目，而不仅仅是控制台应用程序。

前面介绍了创建简单控制台应用程序的所有内容，以及调试它们的各种方法。本书的下一部分将讨论强大的面向对象编程技术。

7.4 练习

(1) “使用 Trace.WriteLine() 要优于使用 Debug.WriteLine()，因为调试版本仅能用于调试程序。”这个观点正确吗？为什么？

(2) 为一个简单的应用程序编写代码，其中包含一个循环，该循环在运行 5000 次后产生一个错误。使用断点在第 5000 次循环出现错误前进入中断模式(注意产生错误的一种简单方式是试图访问一个不存在的数组元素，例如在一个有 100 个元素的数组中，访问 myArray[1000])。

(3) “只有在不执行 catch 块的情况下，才执行 finally 代码块”，对吗？

(4) 下面定义了一个枚举数据类型 orientation。编写一个应用程序，使用结构化异常处理(SEH)把 byte 类型的变量安全地强制转换为 orientation 类型变量。注意，可以使用 checked 关键字强制抛出异常，下面是一个示例。可以在应用程序中使用这段代码：

```
enum orientation : byte
{
    north = 1,
    south = 2,
    east = 3,
    west = 4
}
myDirection = checked((orientation)myByte);
```

附录 A 给出了练习答案。

7.5 本章要点

主 题	重 要 概 念
错误类型	在编译期间和运行期间，致命错误(语法错误)都会使应用程序完全失败，语义错误或逻辑错误比较微妙，可能会使应用程序执行不正确，或者以未预料到的方式执行
输出调试信息	我们可以编写代码，把有帮助的信息输出到 Output 窗口中，以帮助在 IDE 中进行调试。为此需要使用 Debug 和 Trace 系列函数，其中 Debug 函数在发布版本中会被忽略。对于投入生产的应用程序，应把调试输出写入日志文件。在 VS 中，还可以使用跟踪点输出调试信息

(续表)

主 题	重 要 概 念
中断模式	可以通过断点、判定语句, 或者在发生未处理的异常时, 手工进入中断模式(暂停应用程序的状态)。可以在代码的任意位置添加断点, 在 VS 中, 还可以把断点配置为仅在特定条件下中断执行。在中断模式下, 可以检查变量的内容(使用各种调试信息窗口), 每次执行一行代码, 以帮助确定哪里出现了错误
异常	异常是运行期间发生的错误, 可以通过编程方式捕获和处理这种错误, 以防应用程序终止。调用函数或处理变量时, 可能会发生许多不同类型的异常。还可以使用 <code>throw</code> 关键字生成异常
异常处理	代码中未处理的异常会使应用程序终止。使用 <code>try</code> 、 <code>catch</code> 和 <code>finally</code> 代码块处理异常。 <code>Try</code> 块标记了一个启用异常处理的代码段, <code>catch</code> 块包含的代码仅在异常发生时执行, 它可以匹配特定类型的异常, 还可以包含多个 <code>catch</code> 块。 <code>Finally</code> 块指定异常处理完毕后执行的代码, 如果没有发生异常, <code>Finally</code> 块就指定在 <code>try</code> 块执行完毕后执行的代码。只能包含一个 <code>Finally</code> 块, 如果包含了 <code>catch</code> 块, <code>Finally</code> 块就是可选的

## 面向对象编程简介

本章的主要内容:

- 什么是面向对象编程
- OOP 技术
- Windows Forms 应用程序对 OOP 的依赖关系

本书前面介绍了 C#语法和编程的所有基础知识,以及调试应用程序的方法。现在我们已经可以编写出能使用的控制台应用程序了。但是,要了解 C#语言和.NET Framework 的强大功能,还需要使用面向对象编程(Object-Oriented Programming, OOP)技术。实际上,前面已经使用了这些技术,但为了使学习任务简单一些,在列出代码示例时没有重点讲述该技术。

本章先不考虑代码,而主要探讨 OOP 的原理。OOP 会很快把我们领回 C#语言,因为它与 OOP 是一种共生关系。本章介绍的所有概念在后面的章节中都会再次讨论,并用演示性的代码来说明。所以,如果您在第一次阅读本章时没有掌握所有的内容,不必惊慌。

首先介绍 OOP 的基础知识,包括回答最基本的问题“什么是对象?”。很快您就会发现有许多术语与 OOP 有关,这些术语最初很容易混淆,但本章提供了大量的解释。使用 OOP 需要以另一种方式来看待编程。

除了讨论 OOP 的一般原理外,本章还将进入一个对于全面理解 OOP 非常重要的领域:Windows Forms 应用程序。此类应用程序(它们使用 Windows 环境和诸如菜单、按钮等特性)有许多描述性的区域,在 Windows Forms 环境中可以有效地说明 OOP 要点。



本章中的 OOP 实际上是.NET OOP,这里讲述的一些技术不能应用于其他 OOP 环境。在编写 C#程序时,使用的是.NET 特有的 OOP,因此专注于这些方面是明智之举。



## 8.1 面向对象编程的含义

面向对象编程是创建计算机应用程序的一种相当新的方法，它解决了传统编程技巧带来的许多问题。前面介绍的编程方法称为函数(或过程)化编程，常常会导致所谓的单一应用程序，即所有的功能都包含在几个代码模块中(常常是一个代码模块)。而使用OOP技术，常常要使用许多代码模块，每个模块都提供特定的功能，每个模块都是孤立的，甚至与其他模块完全独立。这种模块化编程方法提供了非常大的多样性，大大增加了重用代码的机会。

要进一步说明这个问题，假定计算机上的一个高性能应用程序是一辆一流赛车。如果使用传统的编程技巧，这辆赛车就是一个单元。如果要改进该车，就必须替换整车，把它送回厂商那里，让汽车专家升级它，或者购买一辆新车。如果使用OOP技术，就只需从厂商处购买新的引擎，自己按照其说明替换它，而不必用钢锯切割车体。

在传统的应用程序中，执行流常常是简单的、线性的。把应用程序加载到内存中，从A点开始执行，在B点结束，然后从内存中卸载，在这个过程中可能用到其他各种实体，例如存储介质上的文件或显卡的功能，但处理的主体总是位于一个地方。此时的代码一般与使用各种数学和逻辑方式处理数据相关。处理方法通常比较简单，使用基本的数据类型，例如整型和布尔值，建立比较复杂的数据表达方式。

而使用OOP，事情就不是这么直接了。尽管可以获得相同的效果，但其实现方式是完全不同的。OOP技术以结构、数据的含义以及数据和数据之间的交互操作为基础。这通常意味着要把更多的精力放在项目的设计阶段，但项目的可扩展性比较高。一旦对某种类型的数据的表达方式达成一致，这种表达方式就会应用到应用程序以后的版本中，甚至是全新的应用程序中。这种一致的表达方式可以大大减少开发时间。这就是上述赛车示例的工作原理。这里的一致是“引擎”的代码是结构化的，这样就可以很容易地替换成新代码(即新引擎)，而不需要找厂商帮忙。这也表示，引擎创建出来后可以用于其他目的，可以把它安装到另一辆车上，或者用它驱动潜艇。

除了数据表达方式的一致性外，OOP编程还常常可以简化任务，因为较抽象实体的结构和用法也是一致的。例如，不仅把输出结果发送给设备(如打印机)所使用的数据格式是一致的，而且与该设备交换数据的方法也是一致的，这包括它理解的指令等等。回到赛车的示例上，要达成的一致包括引擎如何连接到油箱上，如何把驱动力传送给车轮等。

顾名思义，OOP技术要使用对象。

### 8.1.1 对象的含义

对象就是OOP应用程序的一个组成部件。这个组成部件封装了部分应用程序，这部分程序可以是一个过程、一些数据或一些更抽象的实体。

简单地说，对象非常类似于本书前面讨论的结构类型，包含变量成员和函数类型。它所包含的变量组成了存储在对象中的数据，其中包含的函数可以访问对象的功能。略为复杂的对象可能不包含任何数据，而只包含函数，表示一个过程。例如，可以使用表示打印机的对象，其中的函数可以控制打印机(允许打印文档、测试页等)。

C#中的对象是从类型中创建的，就像前面的变量一样。对象的类型在OOP中有一个特殊的名称：类。可以使用类的定义实例化对象，这表示创建该类的一个实例。“类的实例”和对象含义相同，注



意“类”和“对象”是完全不同的概念。



术语“类”和“对象”常常混淆，从一开始就正确区分它们是非常重要的，使用前面的赛车示例有助于区分这两个术语。在这个示例中，类是指汽车的模板，或者用于构建汽车的规划。汽车本身是这些规划的实例，所以可以看作对象。

本章将使用统一建模语言(Unified Modeling Language, UML)语法研究类和对象。UML 是为应用程序建模而设计的，从组成应用程序的对象，到它们执行的操作，到我们希望有的用例，应有尽有。这里只使用这个语言的基本部分，在使用它们的过程中进行解释，但不考虑比较复杂的部分，因为 UML 是一个很专业的主题，所以需要整本书的篇幅来讨论。



VS 有一个类查看器，它是一个功能很强大的工具，可用于以类似的方式显示类。但为了简单起见，本章的图是手绘的。

图 8-1 是打印机类 `Printer` 的 UML 表示方法。类名显示在这个框的顶部(后面将论述下面两个区域)。

图 8-2 是这个 `Printer` 类的一个实例 `myPrinter`。

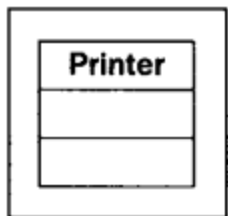


图 8-1

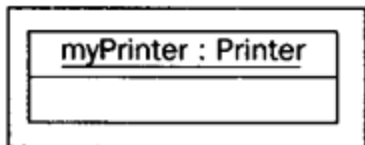


图 8-2

在顶部，实例名显示在前面，后面是类名。这两个名称用一个冒号分隔。

### 1. 属性和字段

可以通过属性和字段访问对象中包含的数据。这个对象数据可以用于区分不同的对象，因为同一个类的不同对象在属性和字段中存储了不同的值。

包含在对象中的不同数据构成了对象的状态。假定一个对象类表示一杯咖啡，叫作 `CupOfCoffee`。在实例化这个类(即创建这个类的对象)时，必须提供对类有意义的状态。此时可以使用属性和字段，让代码能通过该对象设置要使用的咖啡品牌，咖啡中是否加牛奶或方糖，咖啡是否即溶等。于是，给定的这杯咖啡对象就有了指定的状态，例如，加牛奶和两块方糖的哥伦比亚滴滤咖啡。

字段和属性都可以键入，所以可以把信息存储在字段和属性中，作为 `string` 值、`int` 值等。但是，属性与字段是不同的，因为属性不提供对数据的直接访问。对象能让用户不考虑数据的细节，不需要在属性中用一对一的方式表示。如果在 `CupOfCoffee` 实例中使用一个字段表示方糖的数量，用户就可以在该字段中放置自己喜欢的值，其取值范围仅由存储该信息的类型来限制。例如，如果使用 `int` 来存储这个数据，用户就可以使用 `-2 147 483 648~2 147 483 647` 之间的任意值，如第 3 章所述。显然，并不是所有的值都有意义，尤其是负值，一些较大的正值将需要非常大的咖啡杯。但如果使

用一个属性来表示，就可以限制这个值，例如为 0~2 之间的一个数字。

一般情况下，在访问状态时最好提供属性，而不是字段，因为这样可以更好地控制各种行为，这个选择不会影响使用对象实例的代码，因为使用属性和字段的语法是相同的。

对属性的读写访问也可以由对象来明确定义。某些属性是只读的，只能查看它们的值，而不能改变它们(至少不能直接改变)。这常常是同时读取几个状态的一个有效技巧。CupOfCoffee 类有一个只读属性 Description，在请求它时，就返回一个字符串，表示该类的一个实例的状态(例如前面给出的字符串)。也可以通过查看几个属性，把相同的数据组合起来，但这样的属性可以节省时间和精力。还可以有只写的属性，其操作方式是类似的。

除了对属性的读/写访问外，还可以为字段和属性指定另一种访问权限，称为可访问性。这种可访问性确定了什么代码可以访问这些成员，它们是可用于所有的代码(公共)，还是只能用于类中的代码(私有)，或者更复杂的模式(详见本章后面的内容)。常见的情况是把字段设置为私有，通过公共属性访问它们。这样，类中的代码就可以直接访问存储在字段中的数据，而公共属性禁止外部用户访问这些数据，以防他们在其中放置无效的内容。公共成员是类可以访问的成员。

要更清晰地阐明这个问题，可以把可访问性与变量的作用域等同起来。例如，私有字段和属性可以看作是拥有它们的对象的局部成员，而公共字段和属性的作用域也包括对象以外的代码。

在类的 UML 表示方法中，用第二部分显示属性和字段，如图 8-3 所示。

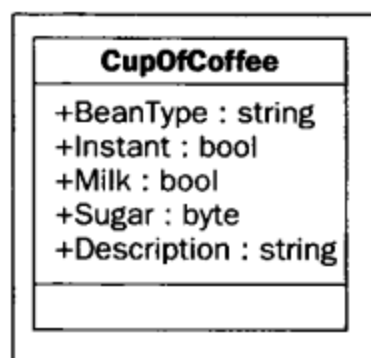


图 8-3

这是 CupOfCoffee 类的表示方式，前面为它定义了 5 个成员(属性或字段，在 UML 中，它们没有区别)。每个成员都包含下述信息：

- 可访问性：+号表示公共成员，-号表示私有成员。但一般情况下，本章的图中不显示私有成员，因为这些信息是类内部的信息。至于读/写访问，则不提供任何信息。
- 成员名。
- 成员的类型。

冒号用于分隔成员名和类型。

## 2. 方法

“方法”这个术语用于表示对象中的函数。这些函数调用的方式与其他函数相同，使用返回值和参数的方式也相同(详见第 6 章)。

方法用于提供访问对象的功能。与字段和属性一样，方法也可以是公共的或私有的，按照需要限制外部代码的访问。它们常常使用对象状态影响它们的操作，在需要时访问私有成员，如私有字段。例如，CupOfCoffee 类定义了一个方法 AddSugar()，该方法对递增方糖数提供了比设置相应的 Sugar 属性更易读的语法。

在 UML 的对象框中，方法显示在第三部分，如图 8-4 所示。

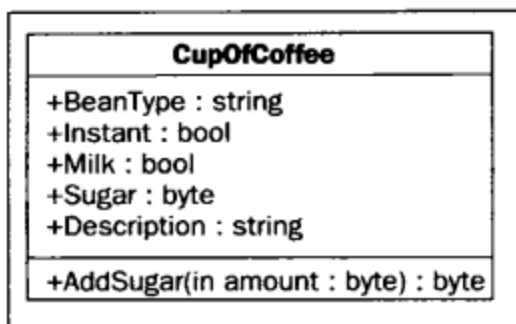


图 8-4

其语法类似于字段和属性，但最后显示的类型是返回类型，在这一部分，还显示了方法的参数。在 UML 中，每个参数都带有下列标识符之一：in、out 或 inout。它们用于表示数据流的方向，其中 out 和 inout 大致对应于第 6 章讨论的 C#关键字 out 和 ref。in 大致对应于 C#中不使用这两个关键字的情形。

### 8.1.2 一切皆对象

本书一直在使用对象、属性和方法。实际上，C#和.NET Framework 中的所有东西都是对象。控制台应用程序中的 Main()函数就是类的一个方法。前面介绍的每个变量类型都是一个类。前面使用的每个命令都是一个属性或方法，例如，<String>.Length 和 <String>.ToUpper()等。句点字符把对象实例名和属性或方法名分隔开来，方法名后面的()把方法与属性区分开来。

对象无处不在，使用它们的语法通常比较简单，这使我们可以集中精力讨论 C#中一些比较基础的方面。从现在开始详细介绍对象。这里讨论的概念都具有深远的影响。它们甚至可以应用到简单的 int 变量上。

### 8.1.3 对象的生命周期

每个对象都有一个明确定义的生命周期，除了“正在使用”的正常状态之外，还有两个重要的阶段：

- **构造阶段：**对象最初进行实例化的时期。这个初始化过程称为构造阶段，由构造函数完成。
- **析构阶段：**在删除一个对象时，常常需要执行一些清理工作，例如，释放内存，这由析构函数完成。

#### 1. 构造函数

对象的初始化过程是自动完成的。我们不需要找一个适于存储新对象的内存空间。但是，在初始化对象的过程中，有时需要执行一些额外的工作。例如，需要初始化对象存储的数据。构造函数就是用于初始化数据的函数。

所有的类定义都至少包含一个构造函数。在这些构造函数中，可能有一个默认的构造函数，该函数没有参数，与类同名。类定义还可能包含几个带有参数的构造函数，称为非默认的构造函数。代码可以使用它们以许多方式实例化对象，例如给存储在对象中的数据提供初始值。

在 C#中，用 new 关键字来调用构造函数。例如，可以用下面的方式通过其默认的构造函数实例化一个 CupOfCoffee 对象：

```
CupOfCoffee myCup = new CupOfCoffee();
```

还可以用非默认的构造函数来创建对象。例如，CupOfCoffee 类有一个非默认的构造函数，它使用一个参数在初始化时设置咖啡豆的品牌：

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

构造函数与字段、属性和方法一样，可以是公共或私有的。在类外部的代码不能使用私有构造函数实例化对象，而必须使用公共构造函数。这样，就可以要求类的用户使用非默认的构造函数(把默认构造函数设置为私有的)。

一些类没有公共的构造函数，外部的代码就不可能实例化它们，这些类称为不可创建的类，但如稍后所述，这些类并不是完全没有用的。

2. 析构函数

.NET Framework 使用析构函数清理对象。一般情况下，不需要提供析构函数的代码，而是由默认的析构函数自动执行操作。但是，如果在删除对象实例前，需要完成一些重要的操作，就应提供特定的析构函数。

例如，如果变量超出了范围，代码就不能访问它，但该变量仍存在于计算机内存的某个地方。只有在.NET 运行程序执行其垃圾回收，进行清理时，该实例才被彻底删除。



不应依赖析构函数释放对象实例使用的资源，因为在不再使用某个对象后，该资源会长时间被该对象占用。如果所使用的资源非常重要，这样做就有可能出问题。有一个解决方法，参阅本章后面的“可删除对象”一节。

8.1.4 静态和实例类成员

属性、方法和字段等成员是对象实例所特有的，此外，还有静态成员(也称为共享成员，尤其是 Visual Basic 用户常常使用这个术语)，例如静态方法、静态属性或静态字段。静态成员可以在类的实例之间共享，所以可以将它们看作是类的全局对象。静态属性和静态字段可以访问独立于任何对象实例的数据，静态方法可以执行与对象类型相关、但与对象实例无关的命令。在使用静态成员时，甚至不需要实例化对象。

例如，前面使用的 Console.WriteLine()和 Convert.ToString()方法就是静态的，根本不需要实例化 Console 或 Convert 类(如果试着进行这样的实例化，操作会失败，因为这些类的构造函数不是可公共访问的，如前所述)。

许多情况下，静态属性和方法有很好的效果。例如，可以使用静态属性跟踪给类创建了多少个实例。在 UML 语法中，类的静态成员用下划线表示，如图 8-5 所示。

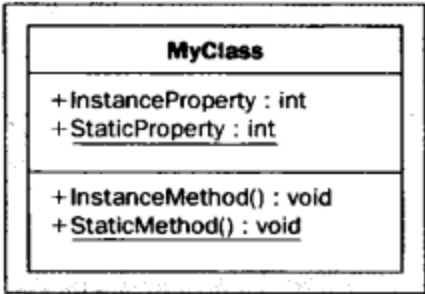


图 8-5

## 1. 静态构造函数

使用类中的静态成员时，需要预先初始化这些成员。在声明时，可以给静态成员提供一个初始值，但有时需要执行更复杂的初始化，或者在赋值、执行静态方法之前执行某些操作。

使用静态构造函数可以执行此类初始化任务。一个类只能有一个静态构造函数，该构造函数不能有访问修饰符，也不能带任何参数。静态构造函数不能直接调用，只能在下述情况下执行：

- 创建包含静态构造函数的类实例时
- 访问包含静态构造函数的类的静态成员时

在这两种情况下，会先调用静态构造函数，之后实例化类或访问静态成员。无论创建了多少个类实例，其静态构造函数都只调用一次。为了区分静态构造函数和本章前面介绍的构造函数，也将所有非静态构造函数称作实例构造函数。

## 2. 静态类

我们常常希望类只包含静态成员，且不能用于实例化对象(如Console)。为此，一种简单的方法是使用静态类，而不是把类的构造函数设置为私有。静态类只能包含静态成员，不需要实例构造函数，因为按照定义，它根本不能实例化。但静态类可以有一个静态构造函数，如上一节所述。



如果以前完全没有接触过 OOP，在阅读本章的其他内容之前，应该停下来将 OOP 研究一番。在学习更复杂的 OOP 内容之前，全面掌握基础知识是很重要的。

## 8.2 OOP 技术

前面介绍了一些基础知识，知道对象是什么，以及对象的工作原理，下面讨论对象的其他一些特性，包括：

- 接口
- 继承
- 多态性
- 对象之间的关系
- 运算符重载
- 事件
- 引用类型和值类型

### 8.2.1 接口

接口是把公共实例(非静态)方法和属性组合起来，以封装特定功能的一个集合。一旦定义了接口，就可以在类中实现它。这样，类就可以支持接口所指定的所有属性和成员。

注意，接口不能单独存在。不能像实例化一个类那样实例化接口。另外，接口不能包含实现其成员的任何代码，而只能定义成员本身。实现过程必须在实现接口的类中完成。

在前面的咖啡示例中，可以把通用属性和方法例如 AddSugar()、Milk、Sugar 和 Instant 组合到



一个接口中，这个接口称为 `IHotDrink`(接口的名称一般用大写字母 I 开头)。然后就可以在其他对象上使用该接口，例如 `CupOfTea` 类的对象。所以可以用类似的方式处理这些对象，而对象仍保有自己的属性(例如 `CupOfCoffee` 仍有属性 `BeanType`，`CupOfTea` 仍有属性 `LeafType`)。

在UML 中，在对象上实现的接口用“棒棒糖”语法来表示。在图 8-6 中，用与类相似的语法把 `IHotDrink` 的成员放在一个单独的框中。

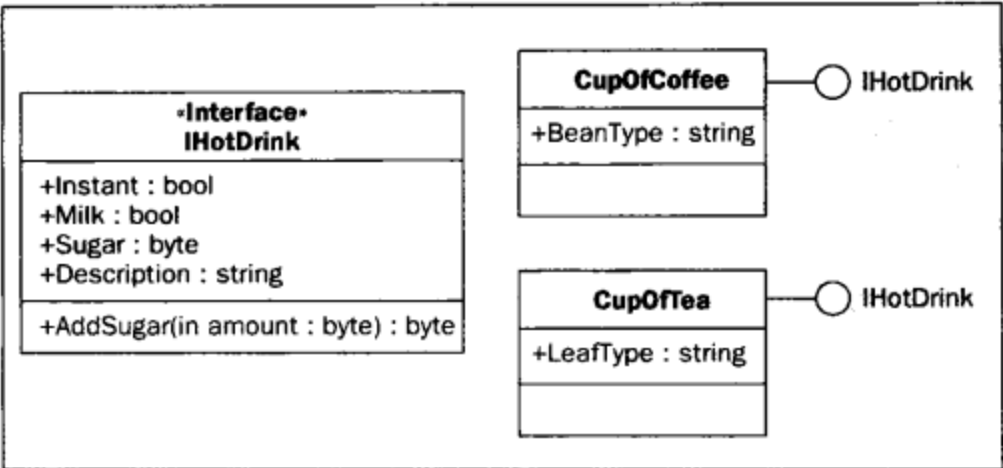


图 8-6

一个类可以支持多个接口，多个类也可以支持相同的接口。所以接口的概念让用户和其他开发人员更容易理解其他人的代码。例如，有一些代码使用一个带某接口的对象。假定不使用这个对象的其他属性和方法，就可以用另一个对象代替这个对象(例如，使用上述 `IHotDrink` 接口的代码可以处理 `CupOfCoffee` 和 `CupOfTea` 实例)。另外，该对象的开发人员可以提供该对象的更新版本，只要它支持已经在用的接口，就可以在代码中使用这个新版本。

在发布接口后，即接口可以用于其他开发人员或终端用户后，最好不要修改它。理解这一点的一种方式是把接口看作类的创建者和使用者之间的契约。“每个支持接口 X 的类都支持这些方法和属性”是有效的。如果以后修改了接口，也许是升级了底层的代码，该接口的使用者就不能正确运行接口，甚至失败。我们应创建一个新的接口，来扩展旧接口，例如包含一个版本号，如 `X2`。这是创建接口的标准方式，以后我们会常常遇到编了号的接口。

可删除的对象

`IDisposable` 接口特别有趣。支持 `IDisposable` 接口的对象必须实现其 `Dispose()` 方法，即它们必须提供这个方法的代码。当不再需要某个对象(例如，在对象超出作用域之前)时，就调用这个方法，释放重要的资源，否则，该资源会等到对垃圾回收调用析构方法时才释放。这样可以更好地控制对象所使用的资源。

C#允许使用一种可以优化使用这个方法的结构。`using` 关键字可以在代码块中初始化使用重要资源的对象，会在这个代码块的末尾自动调用 `Dispose()` 方法，用法如下：

```
<ClassName> <VariableName> = new <ClassName>()
...
using (<VariableName>)
{
    ...
}
```

或者把初始化对象<VariableName>作为 `using` 语句的一部分：



```
using (<ClassName> <VariableName> = new <ClassName>())
{
    ...
}
```

在这两种情况下，可以在 `using` 代码块中使用变量 `<VariableName>`，并在代码块的末尾自动删除(在代码块执行完毕后，调用 `Dispose()`)。

## 8.2.2 继承

继承是 OOP 最重要的特性之一。任何类都可以从另一个类中继承，这就是说，这个类拥有它继承的类的所有成员。在 OOP 中，被继承(也称为派生)的类称为父类(也称为基类)。注意，C# 中的对象仅能直接派生于一个基类，当然基类也可以有自己的基类。

继承性可以从一个较一般的基类扩展或创建更多的特定类。例如，考虑一个代表农场家畜的类(80 多岁的一流开发人员 Old MacDonald 在他的家畜应用程序中使用)。这个类叫作 `Animal`，拥有 `EatFood()` 或 `Breed()` 等方法，我们可以创建一个派生类 `Cow`，支持所有这些方法，它也有自己的方法，如 `Moo()` 和 `SupplyMilk()`。还可以创建另一个派生类 `Chicken`，该类有 `Cluck()` 和 `LayEgg()` 方法。

在 UML 中，用箭头表示继承，如图 8-7 所示。

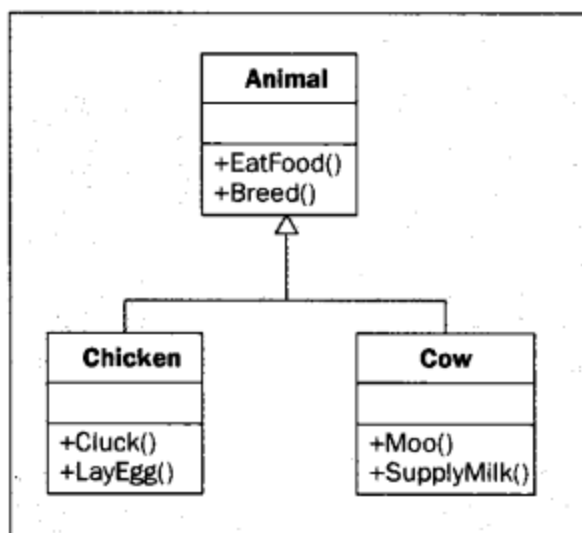


图 8-7



为了简洁起见，图 8-7 中省略了成员的返回类型。

在继承一个基类时，成员的可访问性就成了一个重要的问题。派生类不能访问基类的私有成员，但可以访问其公共成员。不过，派生类和外部的代码都可以访问公共成员。这就是说，只使用这两个可访问性，不能让一个成员可由基类和派生类访问，而不能由外部的代码访问。

为了解决这个问题，C# 提供了第三种可访问性：`protected`，只有派生类才能访问 `protected` 成员。对于外部代码来说，这个可访问性与私有成员一样：外部代码不能访问 `private` 成员和 `protected` 成员。

除了定义成员的保护级别外，我们还可以为成员定义其继承行为。基类的成员可以是虚拟的，也就是说，成员可以由继承它的类重写。派生类可以提供成员的其他实现代码。这种实现代码不会删除原来的代码，仍可以在类中访问原来的代码，但外部代码不能访问它们。如果没有提供其他实现方式，通过派生类使用成员的外部代码就自动访问基类中成员的实现代码。



虚拟成员不能是私有成员，因为这样会自相矛盾——不能说成员可以由派生类重写，同时派生类又不能访问它。

在前面的家畜示例中，可以把 `EatFood()` 变成虚拟成员，在派生类中为它提供新的实现代码，例如为 `Cow` 类提供新实现代码，如图 8-8 所示。这里显示了 `Animal` 和 `Cow` 类的 `EatFood()` 方法，说明它们有自己的实现代码。

基类还可以定义为抽象类。抽象类不能直接实例化。要使用抽象类，必须继承这个类，抽象类可以有抽象成员，这些成员在基类中没有实现代码，这些实现代码必须在派生类中提供。如果 `Animal` 是一个抽象类，UML 就会如图 8-9 所示。



抽象类名以斜体显示(有时它们的方框有一个短横线)。

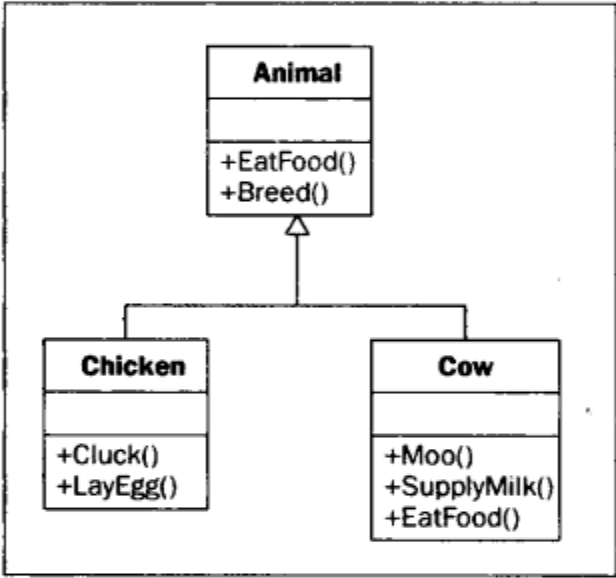


图 8-8

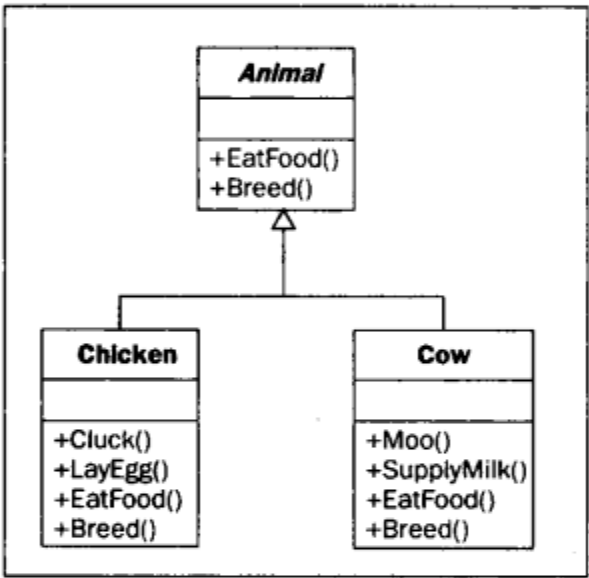


图 8-9

在图 8-9 中，`EatFood()` 和 `Breed()` 都显示在派生类 `Chicken` 和 `Cow` 中，这说明这些方法是抽象的(必须在派生类中重写)或者虚拟的(已经在 `Chicken` 和 `Cow` 中重写)。当然，抽象基类可以提供成员的实现代码，这是很常见的。不能实例化抽象类，并不意味着不能在抽象类中封装功能。

最后，类可以是密封(seal)的。密封的类不能用作基类，所以没有派生类。

在 C# 中，所有的对象都有一个共同的基类 `object`(在 .NET Framework 中，它是 `System.Object` 类的别名)。第 9 章将详细介绍这个类。



如本章前面所述，接口也可以继承自其他接口。与类不同的是，接口可以继承多个基接口(与类可以支持多个接口的方式类似)。

### 8.2.3 多态性

继承的一个结果是派生于基类的类在方法和属性上有一定的重叠,因此,可以使用相同的语法处理从同一个基类实例化的对象。例如,如果基类 `Animal` 有一个 `EatFood()` 方法,则从派生于它的类 `Cow` 和 `Chicken` 中调用这个方法,其语法是类似的:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

多态性则更推进了一步。可以把某个派生类型的变量赋给基本类型的变量,例如:

```
Animal myAnimal = myCow;
```

不需要进行强制类型转换,就可以通过这个变量调用基类的方法:

```
myAnimal.EatFood();
```

结果是调用派生类中的 `EatFood()` 的实现代码。注意,不能以相同的方式调用派生类上定义的方法,下面的代码不能运行:

```
myAnimal.Moo();
```

但是,可以把基本类型的变量转换为派生类变量,调用派生类的方法,如下所示:

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

如果原始变量的类型不是 `Cow` 或派生于 `Cow` 的类型,这个强制类型转换就会引发一个异常。有许多方式说明对象的类型是什么,详见下一章。

在派生于同一个类的不同对象上执行任务时,多态性是一种极有效的技巧,其使用的代码最少。注意并不是只有共享同一个父类的类才能利用多态性。只要子类和孙子类在继承层次结构中有一个相同的类,它们就可以用同样的方式利用多态性。

还要注意,在 C# 中,所有的类都派生于同一个类 `object`, `object` 是继承层次结构中的根。所以可以把所有对象看作是 `object` 类的实例。这就是在建立字符串时, `Console.WriteLine()` 可以处理无数多种参数组合的原因。第一个参数后面的每个参数都可以看作是一个 `object` 实例,所以可以把任何对象的输出结果写到屏幕上。为此,需要调用方法 `ToString()` (`object` 的一个成员),我们可以重写这个方法,为自己的类提供合适的实现代码,或者使用默认实现代码,返回类名(根据它所在的名称空间,返回类的修饰名)。

#### 接口的多态性

尽管不能像对象那样实例化接口,但可以建立接口类型的变量,然后就可以在支持该接口的对象上,使用这个变量访问该接口提供的方法和属性。

例如,假定不使用基类 `Animal` 提供 `EatFood()` 方法,而是把该方法放在 `IConsume` 接口上。`Cow` 和 `Chicken` 类也支持这个接口,唯一的区别是它们必须提供 `EatFood()` 方法的实现代码(因为接口不包含实现代码),接着就可以使用下述代码访问该方法了:

```

Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();

```

这就提供了以相同方式访问多个对象的简单方式，且不依赖于一个公共的基类。例如，这个接口可以由派生于 `Vegetable` 的 `VenusFlyTrap` 类实现，而不是由 `Animal` 实现：

```

VenusFlyTrap myVenusFlyTrap = new VenusFlyTrap();
IConsume consumeInterface;
consumeInterface = myVenusFlyTrap;
consumeInterface.EatFood();

```

在这段代码中，调用 `consumeInterface.EatFood()` 的结果是调用 `Cow`、`Chicken` 或 `VenusFlyTrap` 类的 `EatFood()` 方法，这取决于哪个实例被赋予了接口类型的变量。

注意，派生类会继承其基类支持的接口。在上面的第一个示例中，要么是 `Animal` 支持 `IConsume`，要么是 `Cow` 和 `Chicken` 支持 `IConsume`。有共同基类的类不一定有共同的接口，反之亦然。

## 8.2.4 对象之间的关系

继承是对象之间的一种简单关系，可以让派生类完整地获得基类的特性，而派生类也可以访问基类内部的一些工作代码(通过受保护的成员)。对象之间还有其他一些重要关系。

本节简要讨论下述关系：

- **包含关系：**一个类包含另一个类。这类似于继承关系，但包含类可以控制对被包含类的成员的访问，甚至在使用被包含类的成员前进行其他处理。
- **集合关系：**一个类用作另一个类的多个实例的容器。这类似于对象数组，但集合有其他功能，包括索引、排序和重新设置大小等。

### 1. 包含关系

用一个成员字段包含对象实例，就可以实现包含(containment)关系。这个成员字段可以是公共字段，此时与继承关系一样，容器对象的用户就可以访问它的方法和属性，但不能像继承关系那样，通过派生类访问类的内部代码。

另外，可以让被包含的成员对象变成私有成员。如果这么做，用户就不能直接访问任何成员，即使这些成员是公共的，也不能访问。但可以使用包含类的成员访问这些私有成员。也就是说，可以完全控制被包含的类有什么成员，如果有成员，还可以在访问被包含类的成员前，在包含类的成员上进行其他处理。

例如，`Cow` 类包含一个 `Udder` 类，它有一个公共方法 `Milk()`。`Cow` 对象可以按照要求调用这个方法，作为其 `SupplyMilk()` 方法的一部分，但 `Cow` 对象的用户看不到这些细节。

在 UML 中，被包含类可以用关联线条来表示。对于简单的包含关系，可以用带有 1 的线条说明一对一的关系(一个 `Cow` 实例包含一个 `Udder` 实例)。为清晰起见，也可以把被包含的 `Udder` 类实例表示为 `Cow` 类的私有字段，如图 8-10 所示。

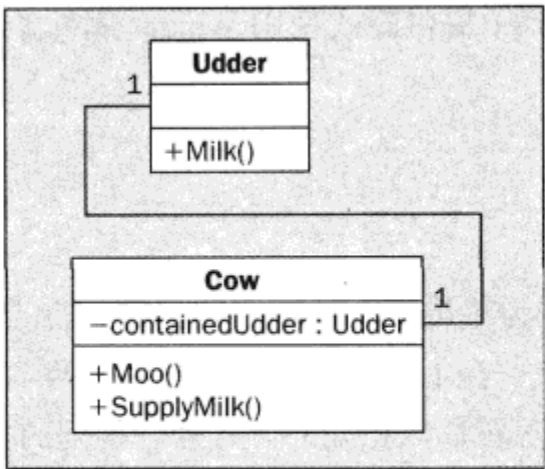


图 8-10

2. 集合关系

第 5 章讨论了如何使用数组存储多个同类变量。这也适用于对象(前面使用的变量类型实际上是对象)。例如：

```
Animal[] animals = new Animal[5];
```

集合基本上是数组，集合以与其他对象相同的方式实现为类。它们通常以所存储的对象名称的复数形式来命名，例如类 `Animals` 就包含 `Animal` 对象的一个集合。

数组与集合的主要区别是，集合通常实现额外的功能，例如 `Add()`和 `Remove()`方法可添加和删除集合中的项。而集合通常有一个 `Item` 属性，它根据对象的索引返回该对象。不仅如此，这个属性还允许以更复杂的访问方式来实现。例如，可以设计一个 `Animals`，让 `Animal` 对象根据其名称来访问。

在 UML 中，这用图 8-11 来表示。

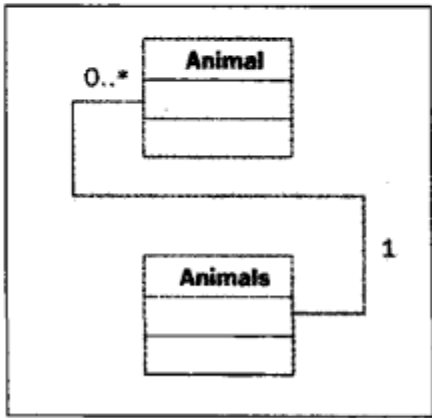


图 8-11

这里遗漏了成员，因为这里描述的是关系。连接线末尾的数字表示一个 `Animals` 对象可以包含 0 个或多个 `Animal` 对象。第 11 章将详细论述集合。

8.2.5 运算符重载

本书前面介绍了如何使用运算符处理简单的变量类型。有时也可以把运算符用于从类实例化而来的对象，因为类可以包含如何处理运算符的指令。

例如，给 `Animal` 添加一个新属性 `Weight`。接着使用下述代码比较家畜的体重：

```
if (cowA.Weight > cowB.Weight)
{
    ...
}
```

使用运算符重载，可以在代码中提供隐式使用 `Weight` 属性的逻辑，如下面的代码所示：

```
if (cowA > cowB)
{
    ...
}
```

大于运算符`>`被重载了。我们为重载运算符编写代码，执行上述操作，这段代码用作类定义的一部分，而该运算符作用于这个类。在上面的示例中，使用了两个 `Cow` 对象，所以运算符重载定义包含在 `Cow` 类中。也可以重载运算符，以相同的方式处理不同的类，其中一个(或两个)类定义包含达到这一目的的代码。

注意，只能采用这种方式重载现有的 C# 运算符，不能创建新的运算符。但是，可以为一元和二元运算符(如`+`)提供实现代码。详见第 13 章。

### 8.2.6 事件

对象可以激活事件，作为它们处理的一部分。事件是非常重要的，可以在代码的其他部分起作用，类似于异常(但功能更强大)。例如，可以在把 `Animal` 对象添加到 `Animals` 集合中时，执行特定的代码，而这部分代码不是 `Animals` 类的一部分，也不是调用 `Add()` 方法的代码的一部分。为此，需要给代码添加事件处理程序，这是一种特殊类型的函数，在事件发生时调用。还需要配置这个处理程序，以监听自己感兴趣的事件。

使用事件可以创建事件驱动的应用程序，这类应用程序比读者此时所能想到的多得多。例如，许多基于 `Windows` 的应用程序完全依赖于事件。每个按钮单击或滚动条拖动操作都是通过事件处理实现的，其中事件是通过鼠标或键盘触发的。

本章的后面将介绍在 `Windows` 应用程序中事件的工作原理，第 13 章将深入讨论事件。

### 8.2.7 引用类型和值类型

在 C# 中，数据根据变量的类型以两种方式中的一种存储在一个变量中。变量的类型分为两种：引用类型和值类型，其区别如下：

- 值类型在内存的一个地方存储它们自己和它们的内容。
- 引用类型存储指向内存中其他某个位置(称为堆)的引用，而在另一个位置存储内容。

实际上，在使用 C# 时，不必过多地考虑这个问题。到目前为止，所使用的 `string` 变量(这是引用类型)与使用其他简单变量(大多数是值类型，例如 `int`)的方式完全相同。

值类型和引用类型的一个主要区别是：值类型总是包含一个值，而引用类型可以是 `null`，表示它们不包含值。但是，可以使用可空类型(这是泛型的一种形式)创建一个值类型，使值类型在这个方面的行为方式类似于引用类型(即可以为 `null`)。这是一个高级论题，详见第 12 章。

只有 `string` 和 `object` 简单类型是引用类型，但数组也是隐式的引用类型。我们创建的每个类都是引用类型，这就是在这里说明这一点的原因。



结构类型和类的重要区别是，结构类型是值类型。您可能认为结构类型和类非常类似，特别是第 6 章介绍了如何在结构类型上使用函数。第 9 章还将进一步探讨这个问题。



### 8.3 Windows 应用程序中的 OOP

第 2 章介绍了如何在 C# 中创建简单的 Windows 应用程序。Windows 应用程序非常依赖 OOP 技术，本节将论述 OOP 技术，说明本章的一些论点。下面通过一个简单示例加以说明。

试一试：使用对象

- (1) 在 C:\BegVCSharp\Chapter08 目录中创建一个新的 Windows 应用程序 Ch08Ex01。
- (2) 使用 Toolbox 添加一个新的按钮控件，使之位于 Form1 的中央，如图 8-12 所示。
- (3) 双击按钮，为鼠标单击事件添加代码，修改代码，如下所示：



```
private void button1_Click(object sender, System.EventArgs e)
{
    ((Button)sender).Text = "Clicked!";
    Button newButton = new Button();
    newButton.Text = "New Button!";
    newButton.Click += new EventHandler(newButton_Click);
    Controls.Add(newButton);
}

private void newButton_Click(object sender, System.EventArgs e)
{
    ((Button)sender).Text = "Clicked!!";
}

}
```

代码段 Ch08Ex01"Form1.cs

- (4) 运行应用程序，窗体如图 8-13 所示。

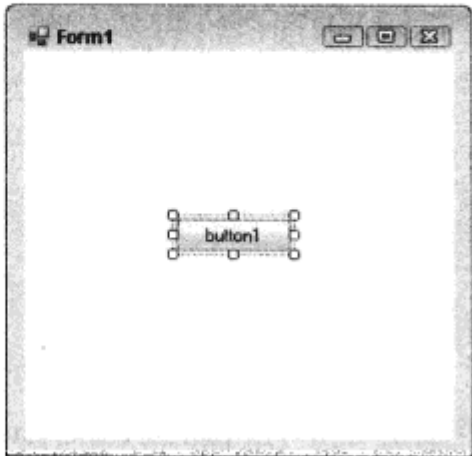


图 8-12



图 8-13

- (5) 单击标记为 button1 的按钮，显示内容将随之变化，如图 8-14 所示。
- (6) 单击标记为 New Button! 的按钮，显示内容将随之变化，如图 8-15 所示。



图 8-14

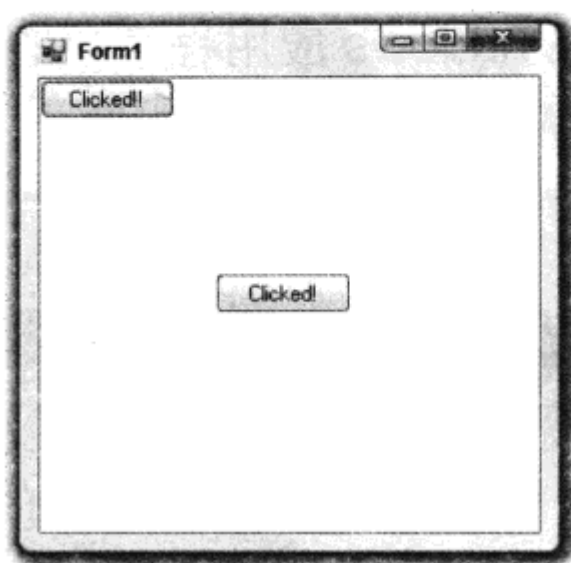


图 8-15

### 示例的说明

添加几行代码，就创建了一个可以完成某项任务的 Windows 应用程序。下面说明 C#中的一些 OOP 技术。即使在谈到 Windows 应用程序时，“一切皆对象”这句话也是正确的。从运行的窗体，到窗体上的控件，都需要使用 OOP 技术。在这个示例中，重点说明本章前面介绍的一些概念，解释如何把它们组合在一起。

在应用程序中，首先是在 Form1 窗体上添加一个新按钮，这个按钮是一个对象，它是 Button 类的一个实例；窗体是 Form1 类的实例，该类从 Form 类派生而来。接着双击按钮，添加一个事件处理程序，监听 Button 类提供的 Click 事件。这个事件处理程序添加到封装应用程序的 Form 对象代码中，是一个私有方法：

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

这段代码使用 C#关键字 `private` 作为修饰符。现在不要考虑这个关键字，第 9 章将详细解释本章提及的 OOP 技术。

我们添加的第一行代码改变了按钮上的文本。它利用了本章前面讨论的多态性。表示按钮的 Button 对象作为一个 `object` 参数发送给事件处理程序，该事件处理程序把参数强制转换为 Button 类型(这是可能的，因为 Button 对象继承于 `System.Object`，`System.Object` 是一个 .NET 类，`object` 是其别名)。然后修改对象的 `Text` 属性，改变显示的文本：

```
((Button)sender).Text = "Clicked!";
```

接着用 `new` 关键字创建一个新 Button 对象(注意在这个项目中设置了名称空间，因此可以使用这个简单的语法，否则，就需要使用这个对象的完整限定名 `System.Windows.Forms.Button`)：

```
Button newButton = new Button();
newButton.Text = "New Button!";
```

在代码的其他地方添加一个新的事件处理程序，以响应新按钮生成的 Click 事件：

```
private void newButton_Click(object sender, System.EventArgs e)
{
}
```

```
((Button)sender).Text = "Clicked!!";
}
```

接着使用一些重载运算符语法,把这个事件处理程序注册为 Click 事件的监听程序。同时使用非默认的构造函数创建一个新的 EventHandler 对象,其名称是新事件处理函数的名称:

```
newButton.Click += new EventHandler(newButton_Click);
```

最后,利用窗体的 Controls 属性,这个属性是一个对象,是窗体上所有控件的集合,通过它的 Add()方法把新按钮添加到窗体上:

```
Controls.Add(newButton);
```

Controls 属性说明,属性不一定是字符串或整型等简单类型,可以是任何类型的对象。这个简短示例几乎使用了本章介绍的所有技术。可以看出,OOP 编程并不复杂——只需要从另一个角度来看待它即可。

## 8.4 小结

本章完整地描述了面向对象技术。我们在 C#编程环境中进行论述,但主要是用示例来说明。本章介绍的 OOP 大都适用于任何语言。

首先介绍基础知识,例如术语“对象”的含义,对象如何成为类的实例。接着讨论对象有各种成员,例如字段、属性和方法。这些成员的可访问性都有一定的限制,然后解释了公共和私有成员。之后,说明成员也可以是受保护的,还可以是虚拟和抽象的(抽象方法只能存在于抽象类中),另外还解释了静态(共享)和实例成员的区别,说明使用静态类的原因。

接下来简要介绍了对象的生命周期,包括如何使用构造函数创建对象,如何使用析构函数删除对象。在说明了接口如何组合对象后,介绍了更高级的对象删除方式:支持 IDisposable 接口的可删除对象。

本章的其他部分重点介绍了 OOP 的特性,其中有许多特性将在随后的章节中详细讨论。我们论述了继承(类可以继承基类),两个版本的多态性(即基类和共享接口),对象如何用于包含一个或多个其他对象(通过包含和集合关系)。最后介绍运算符重载如何用于简化使用对象的语法,对象如何引发事件。

本章的最后部分用一个 Windows 应用程序示例演示了许多理论。第 9 章将介绍如何使用 C# 定义类。

## 8.5 练习

- (1) 下述哪些项在 OOP 中有真实级别的可访问性?
  - 友元
  - 公共
  - 安全

- 私有
  - 受保护的
  - 松散的
  - 通配符
- (2) “必须手动调用对象的析构函数，否则就会浪费资源”的说法正确吗？
- (3) 只有创建一个对象，才能调用其类的静态方法吗？
- (4) 为下述类和接口绘制一个类似于本章介绍的图形的 UML 图：
- 抽象类 HotDrink，它有方法 Drink()、AddMilk()和 AddSugar()，以及属性 Milk 和 Sugar。
  - 接口 ICup，它有方法 Refill()和 Wash()，以及属性 Color 和 Volume。
  - 派生于 HotDrink 的类 CupOfCoffee 支持 ICup 接口，还有一个属性 BeanType。
  - 派生于 HotDrink 的类 CupOfTea 支持 ICup 接口，还有一个属性 LeafType。
- (5) 为一个函数编写一些代码，接受上述示例的两个杯子对象中的任意一个，作为一个参数。该函数应可以为它传送的任何杯子对象调用 AddMilk()、Drink()和 Wash()方法。
- 附录 A 给出了练习答案。

8.6 本章要点

主 题	重 要 概 念
对象和类	对象是 OOP 应用程序的组成部件。类是用于实例化对象的类型定义。对象可以包含数据，提供其他代码可以使用的操作。数据可以通过属性供外部代码使用，操作可以通过方法供外部代码使用。属性和方法都称为类成员。属性可以进行读取访问、写入访问或读写访问。类成员可以是公共的(可用于所有的代码)或私有的(只有类定义中的代码可以使用)。在.NET 中，所有的东西都是对象
对象的生存周期	对象通过调用它的一个构造函数来实例化。不再需要对象时，就执行其析构函数，以删除它。要清理对象，常常需要手工删除它
静态和实例成员	实例成员只能在类的对象实例上使用，静态成员只能直接通过类定义使用，它不与实例关联
接口	接口是可以在类上实现的公共属性和方法的集合。可以给实例类型的变量赋予其类定义实现了该接口的任意对象的值。之后通过该变量，可以使用该接口定义的成员
继承	继承是一个类定义派生于另一个类定义的机制。类从其父类中继承成员，每个类都只能有一个父类。子类不能访问父类的私有成员，但可以定义受保护的成员，受保护的成员只能在该类和派生于该类的子类中使用。子类可以重写父类中定义为虚拟的成员。所有的类都有一个以 System.Object 结尾的继承链，在 C#中，System.Object 有一个别名 Object
多态性	从一个派生类中实例化的所有对象都可以看作是其父类的实例
对象关系和特性	对象可以包含其他对象，也可以表示其他对象的集合。要在表达式中处理对象，常常需要通过运算符重载，定义运算符如何处理对象。对象可以提供事件，事件因某种内部处理而被触发，客户代码可以提供事件处理程序，来响应事件

# 第 9 章

## 定 义 类

### 本章内容:

- 如何在 C# 中定义类和接口
- 如何使用控制可访问性和继承的关键字
- System.Object 类及其在类定义中的作用
- 如何使用 VS 和 VCE 提供的一些帮助工具
- 如何定义类库
- 接口和抽象类的异同
- 结构类型的更多内容
- 复制对象的一些重要信息

第 8 章介绍了面向对象编程(OOP)的特性,本章则要将理论付诸实践,看看如何在 C# 中定义类。本章并不讨论如何定义类的成员,而重点讨论如何定义类本身。这听起来有一定的限制,但不必担心,本章有足够丰富的内容供读者学习。

首先看看基本的类定义语法、用于确定类可访问性的关键字以及指定继承的方式。我们还将介绍接口的定义,因为它们在许多方面都类似于类的定义。

本章的其他部分介绍在 C# 中定义类时涉及到的其他主题。

### 9.1 C# 中的类定义

C# 使用 `class` 关键字来定义类:

```
class MyClass
{
    // Class members.
}
```

这段代码定义了一个类 `MyClass`。定义了一个类后,就可以在项目中能访问该定义的其他位置

对该类进行实例化。默认情况下，类声明为内部的，即只有当前项目中的代码才能访问它。可以使用 `internal` 访问修饰符关键字显式指定，如下所示(但这是不必要的)：

```
internal class MyClass
{
    // Class members.
}
```

另外，还可以指定类是公共的，应该可以由其他项目中的代码来访问。为此，要使用关键字 `public`。

```
public class MyClass
{
    // Class members.
}
```



以这种方式声明的类不能是私有或受保护的。可以把这些声明类的修饰符用于声明类成员，详见第 10 章。

除了这两个访问修饰符关键字外，还可以指定类是抽象的(不能实例化，只能继承，可以有抽象成员)或密封的(`sealed`，不能继承)。为此，可以使用两个互斥的关键字 `abstract` 或 `sealed`。所以，抽象类必须用下述方式声明：

```
public abstract class MyClass
{
    // Class members, may be abstract.
}
```

其中 `MyClass` 是一个公共抽象类，也可以是内部抽象类。

密封类的声明如下所示：

```
public sealed class MyClass
{
    // Class members.
}
```

与抽象类一样，密封类也可以是公共或内部的。

还可以在类定义中指定继承。为此，要在类名的后面加上一个冒号，其后是基类名，例如：

```
public class MyClass : MyBase
{
    // Class members.
}
```

注意，在 C# 的类定义中，只能有一个基类，如果继承了一个抽象类，就必须实现所继承的所有抽象成员(除非派生类也是抽象的)。

编译器不允许派生类的可访问性高于基类。也就是说，内部类可以继承于一个公共基类，但公共类不能继承于一个内部类。因此，下述代码是合法的：

```
public class MyBase
```



```

{
    // Class members.
}

internal class MyClass : MyBase
{
    // Class members.
}

```

但下述代码不能编译:

```

internal class MyBase
{
    // Class members.
}

public class MyClass : MyBase
{
    // Class members.
}

```

如果没有使用基类, 则被定义的类就只继承于基类 `System.Object`(它在C#中的别名是 `object`)。毕竟, 在继承层次结构中, 所有类的根都是 `System.Object`, 稍后将详细介绍这个基类。

除了以这种方式指定基类外, 还可以在冒号之后指定支持的接口。如果指定了基类, 它必须紧跟在冒号的后面, 之后才是指定的接口。如果没有指定基类, 则接口就紧跟在冒号的后面。必须使用逗号分隔基类名(如果有基类)和接口名。

例如, 给 `MyClass` 添加一个接口, 如下所示:

```

public class MyClass : IMyInterface
{
    // Class members.
}

```

所有接口成员都必须在支持该接口的类中实现, 但如果不使用给定的接口成员, 就可以提供一个“空”的实现方式(没有函数代码)。还可以把接口成员实现为抽象类中的抽象成员。

下面的声明是无效的, 因为基类 `MyBase` 不是继承列表中的第一项:

```

public class MyClass : IMyInterface, MyBase
{
    // Class members.
}

```

指定基类和接口的正确方式如下:

```

public class MyClass : MyBase, IMyInterface
{
    // Class members.
}

```

可以指定多个接口, 所以下列代码是有效的:

```

public class MyClass : MyBase, IMyInterface, IMySecondInterface
{

```

```
// Class members.  
}
```

表 9-1 是类定义中可以使用的访问修饰符的组合。

表 9-1

修 饰 符	含 义
无或 internal	只能在当前项目中访问类
public	可以在任何地方访问类
abstract 或 internal abstract	类只能在当前项目中访问，不能实例化，只能供继承之用
public abstract	类可以在任何地方访问，不能实例化，只能供继承之用
sealed 或 internal sealed	类只能在当前项目中访问，不能供派生之用，只能实例化
public sealed	类可以在任何地方访问，不能供派生之用，只能实例化

接口的定义

声明接口的方式与声明类的方式相似，但使用的关键字是 interface，而不是 class，例如：

```
interface IMyInterface  
{  
    // Interface members.  
}
```

访问修饰符关键字 public 和 internal 的使用方式是相同的，与类一样，接口也默认定义为内部接口。所以要使接口可以公开访问，必须使用 public 关键字：

```
public interface IMyInterface  
{  
    // Interface members.  
}
```

不能在接口中使用关键字 abstract 和 sealed，因为这两个修饰符在接口定义中是没有意义的(它们不包含实现代码，所以不能直接实例化，且必须是可以继承的)。

接口的继承也可以用与类继承类似的方式来指定。主要的区别是可以使用多个基接口，例如：

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInterface2  
{  
    // Interface members.  
}
```

接口不是类，所以没有继承 System.Object。但是为了方便起见，System.Object 的成员可以通过接口类型的变量来访问。如上所述，不能用实例化类的方式来实例化接口。下面的示例提供了一些类定义的代码和使用它们的代码。

试一试：定义类

- (1) 在 C:\BegVCSharp\Chapter09 目录中创建一个新的控制台应用程序 Ch09Ex01。
- (2) 修改 Program.cs 中的代码，如下所示：



namespace Ch09Ex01

```

{
    public abstract class MyBase
    {
    }

    internal class MyClass : MyBase
    {
    }

    public interface IMyBaseInterface
    {
    }

    internal interface IMyBaseInterface2
    {
    }

    internal interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
    {
    }

    internal sealed class MyComplexClass : MyClass, IMyInterface
    {
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyComplexClass myObj = new MyComplexClass();
            Console.WriteLine(myObj.ToString());
            Console.ReadKey();
        }
    }
}

```

代码段 Ch09Ex01\Program.cs

(3) 执行项目，结果如图 9-1 所示。

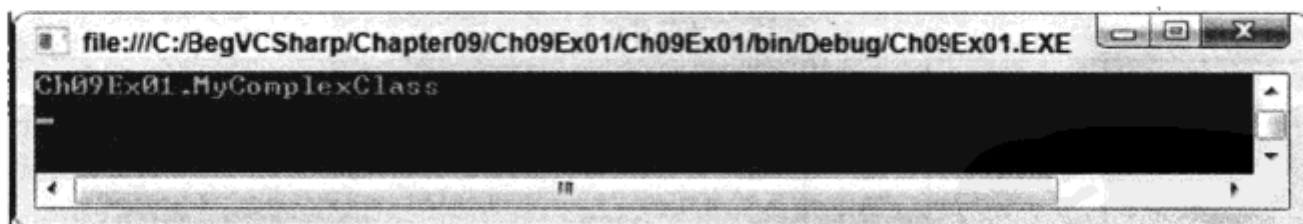


图 9-1

#### 示例的说明

这个项目在下面的继承层次结构中定义了类和接口，如图 9-2 所示。

这里包含 **Program**，是因为这个类的定义方式与其他类的定义方式相同，而它不是主要类层次结构的一部分。这个类处理的 **Main()** 方法是应用程序的入口点。

**MyBase** 和 **IMyBaseInterface** 被定义为公共的，所以它们可以在其他项目中使用。其他类和接口都是内部的，只能在本项目中使用。

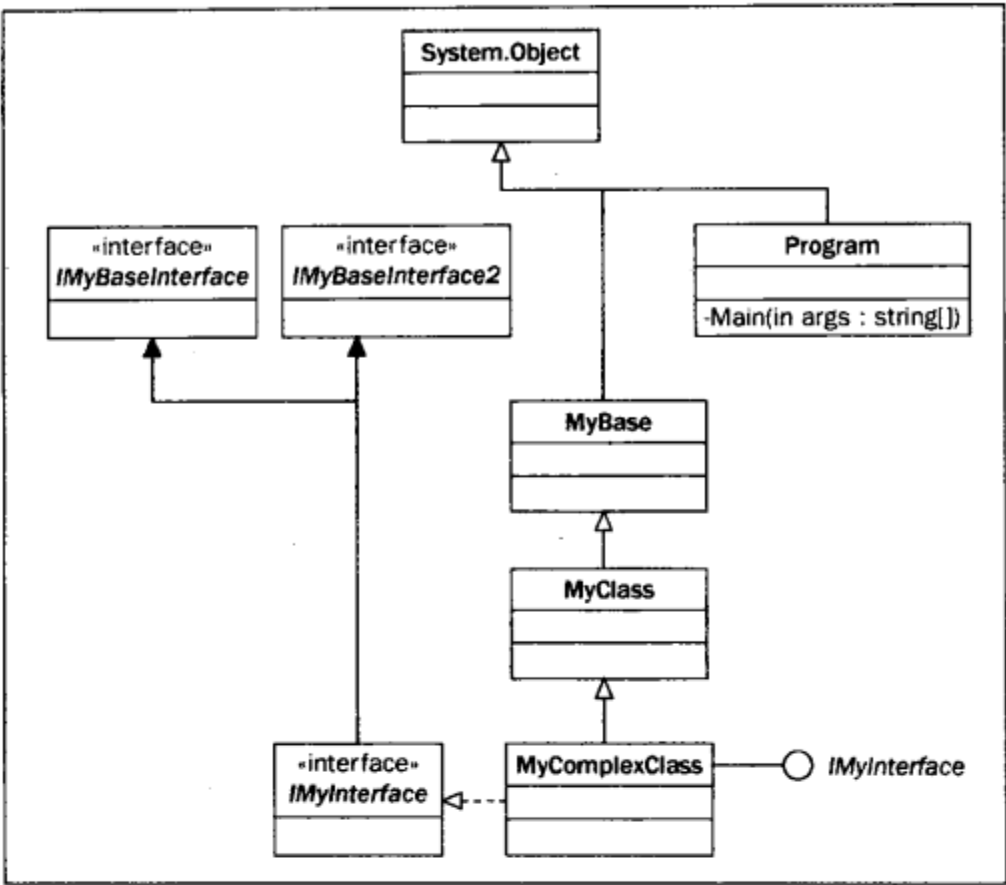


图 9-2

Main()中的代码调用 MyComplexClass 的一个实例 myObj 的 ToString()方法:

```
MyComplexClass myObj = new MyComplexClass();
Console.WriteLine(myObj.ToString());
```

这是继承自System.Object 的一个方法(图中没有显示, 该图省略了这个类的成员, 使图变得更清晰), 并把对象的类名作为一个字符串返回, 该类名用任意相关的命名空间来限定。

这个示例没有完成什么具体的工作, 但本章后面还要利用这个示例演示几个重要概念和技术。

## 9.2 System.Object

因为所有的类都继承于 System.Object, 所以这些类都可以访问该类中受保护的成员和公共的成员。下面看看可供使用的成员有哪些。System.Object 包含的方法如表 9-2 所示。

表 9-2

方 法	返回类型	虚拟	静态	说 明
Object()	N/A	无	无	System.Object 类型的构造函数, 由派生类型的构造函数自动调用
~Object()( 也 称 为 Finalize(), 参见下一节)	N/A	无	无	System.Object 类型的析构函数, 由派生类型的析构函数自动调用, 不能手动调用
Equals(object)	bool	有	无	把调用该方法的对象与另一个对象相比, 如果它们相等, 就返回 true。默认的实现代码会查看对象的参数是否引用了同一个对象(因为对象是引用类型)。如果想以不同的方式来比较对象, 则可以重写该方法, 例如, 比较两个对象的状态

(续表)

方 法	返回类型	虚拟	静态	说 明
Equals (object, object)	bool	无	有	这个方法比较传送给它的两个对象，看看它们是否相等。检查时使用了 Equals(object) 方法。注意，如果两个对象都是空引用，这个方法就返回 true
ReferenceEquals (object,object)	bool	无	有	这个方法比较传送给它的两个对象，看看它们是否是同一个实例的引用
ToString()	String	有	无	返回一个对应于对象实例的字符串。默认情况下，这是一个类类型的限定名称，但可以重写它，给类型提供合适的实现方式
MemberwiseClone()	object	无	无	通过创建一个新对象实例并复制成员，以复制该对象。成员拷贝不会得到这些成员的新实例。新对象的任何引用类型成员都将引用与源类相同的对象，这个方法是受保护的，所以只能在类或派生的类中使用
GetType()	System. Type	无	无	以 System.Type 对象的形式返回对象的类型
GetHashCode()	int	有	无	用作对象的散列函数，这是一个必选函数，返回一个以压缩形式标识的对象状态的值

这些方法是.NET Framework 中对象类型必须支持的基本方法，但我们可能从不使用其中某些类型(或者只在特殊情况下使用，如 GetHashCode())。

在利用多态性时，GetType()是一个有用的方法，允许根据对象的类型来执行不同的操作，而不是像通常那样，对所有的对象都执行相同的操作。例如，如果函数接受一个 object 类型的参数(表示可以给该函数传送任何信息)，就可以在遇到某些对象时执行额外的任务。联合使用 GetType()和 typeof(这是一个 C#运算符，可以把类名转换为 System.Type 对象)，就可以进行比较，如下所示：

```
if (myObj.GetType() == typeof(MyComplexClass))
{
    // myObj is an instance of the class MyComplexClass.
}
```

返回的 System.Type 对象可以做更多的工作，这里不讨论它们。重写 ToString()方法也是非常有效的，特别是在对象的内容中可以用一个人们能理解的字符串表示时，就更是如此。后面的章节将反复讨论这些 System.Object 方法，现在就讨论到这里，后面在需要时再详细讨论。

9.3 构造函数和析构函数

在 C#中定义类时，常常不需要定义相关的构造函数和析构函数，因为在建立代码时，如果没有提供它们，编译器会自动添加它们。但是，如果需要，可以提供自己的构造函数和析构函数，以便

初始化对象和清理对象。

使用下述语法可以把一个简单的构造函数添加到类中：

```
class MyClass
{
    public MyClass()
    {
        // Constructor code.
    }
}
```

这个构造函数与包含它的类同名，且没有参数(使之成为类的默认构造函数)，这是一个公共函数，所以类的对象可以使用这个构造函数进行实例化(详见第 8 章)。

也可以使用私有的默认构造函数，即不能用这个构造函数来创建这个类的对象实例(它是不可创建的，详见第 8 章)：

```
class MyClass
{
    private MyClass()
    {
        // Constructor code.
    }
}
```

最后，也可以用相同的方式给类添加非默认的构造函数，其方法是提供参数，例如：

```
class MyClass
{
    public MyClass()
    {
        // Default constructor code.
    }

    public MyClass(int myint)
    {
        // Nondefault constructor code (uses myInt).
    }
}
```

可提供的构造函数的数量不受限制(当然不能耗尽内存，也不能有相同的参数集，所以“几乎无限制”更合适)。

使用略微不同的语法来声明析构函数。在.NET 中使用的析构函数(由 System.Object 类提供)叫作 Finalize()，但这不是我们用于声明析构函数的名称。使用下面的代码，而不是重写 Finalize()：

```
class MyClass
{
    ~MyClass()
    {
        // Destructor body.
    }
}
```



类的析构函数由带有~前缀的类名(与构造函数的相同)来声明。当进行垃圾回收时,就执行析构函数中的代码,释放资源。在调用这个析构函数后,还将隐式地调用基类的析构函数,包括 System.Object 根类中的 Finalize() 调用。这个技术可以让 .NET Framework 确保调用 Finalize(), 因为重写 Finalize() 是指基类调用需要显式地执行,这是具有潜在危险的(第 10 章将详细讨论如何调用基类的方法)。

## 构造函数的执行序列

如果在类的构造函数中执行多个任务,把这些代码放在一个地方是非常方便的,这与第 6 章论述的把代码放在函数中有相同的优势。使用一个方法就可以把代码放在一个地方(详见第 10 章),而 C# 提供了一个更好的替代方式。任何构造函数都可以配置,在执行自己的代码前调用其他构造函数。

在讨论构造函数前,先看看在默认情况下,创建类的实例时会发生什么情况。除了前面说过的便于把初始化代码集中起来之外,还要了解这些代码。在开发过程中,对象常常并没有按照预期的那样执行,而是在调用构造函数时出现错误。这常常是因为类继承结构中的某个基类没有正确实例化,或者没有正确地给基类构造函数提供信息。如果理解在对象生命周期的这个阶段发生的事情,将更利于解决这类问题。

为了实例化派生的类,必须实例化它的基类。而要实例化这个基类,又必须实例化这个基类的基类,这样一直到实例化 System.Object(所有类的根)为止。结果是无论使用什么构造函数实例化一个类,总是要先调用 System.Object.Object。

无论在派生类上使用什么构造函数(默认的构造函数或非默认的构造函数),除非明确指定,否则就使用基类的默认构造函数(稍后将介绍如何改变这个操作)。下面介绍一个简短示例,说明执行的顺序。考虑下面的对象层次结构:

```
public class MyBaseClass
{
    public MyBaseClass()
    {
    }

    public MyBaseClass(int i)
    {
    }
}

public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass()
    {
    }

    public MyDerivedClass(int i)
    {
    }

    public MyDerivedClass(int i, int j)
    {
    }
```

```
}  
}
```

如果以下面的方式实例化 `MyDerivedClass`:

```
MyDerivedClass myObj = new MyDerivedClass();
```

则执行的顺序如下:

- 执行 `System.Object.Object()`构造函数。
- 执行 `MyBaseClass.MyBaseClass()`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass()`构造函数。

另外, 如果使用下面的语句:

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

则执行的顺序如下:

- 执行 `System.Object.Object()`构造函数。
- 执行 `MyBaseClass.MyBaseClass()`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass(int i)`构造函数。

最后, 如果使用下面的语句:

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

则执行顺序如下:

- 执行 `System.Object.Object()`构造函数。
- 执行 `MyBaseClass.MyBaseClass()`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass(int i, int j)`构造函数。

大多数情况下, 这个系统会正常工作。但是, 有时需要对发生的事件进行更多的控制。例如, 在上面的实例化示例中, 执行的顺序如下:

- 执行 `System.Object.Object()`构造函数。
- 执行 `MyBaseClass.MyBaseClass(int i)`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass(int i, int j)`构造函数。

使用这个顺序可以把使用 `int i` 参数的代码放在 `MyBaseClass(int i)`中, 即 `MyDerivedClass(int i, int j)`构造函数要做的工作比较少, 只需要处理 `int j` 参数(假定 `int i` 参数在两种情况下含义相同, 虽然事情并非总是如此, 但实际上我们常常做这样的安排)。只要愿意, C#就可以指定这种操作。

为此, 只需使用构造函数初始化器, 它把代码放在方法定义的冒号后面。例如, 可以在派生类的构造函数定义中指定所使用的基类构造函数, 如下所示:

```
public class MyDerivedClass : MyBaseClass  
{  
    ...  
  
    public MyDerivedClass(int i, int j) : base(i)  
    {  
    }  
}
```

其中, `base` 关键字指定.NET 实例化过程使用基类中有指定参数的构造函数。这里使用了一个 `int` 参数(其值通过参数 `i` 传送给 `MyDerivedClass` 构造函数), 所以应使用 `MyBaseClass(int i)`。这么做将不调用 `MyBaseClass()`, 而是执行本例前面列出的事件序列——也就是我们希望执行的事件序列。

也可以使用这个关键字指定基类构造函数的字面值, 例如, 使用 `MyDerivedClass` 的默认构造函数调用 `MyBaseClass` 非默认的构造函数:

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass() : base(5)
    {
    }
    ...
}
```

这段代码将执行下述序列:

- 执行 `System.Object.Object()`构造函数。
- 执行 `MyBaseClass.MyBaseClass(int i)`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass()`构造函数。

除了 `base` 关键字外, 这里还可以将另一个关键字 `this` 用作构造函数初始化器。这个关键字指定在调用指定的构造函数前, .NET 实例化过程对当前类使用非默认的构造函数。例如:

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass() : this(5, 6)
    {
    }
    ...

    public MyDerivedClass(int i, int j) : base(i)
    {
    }
}
```

这段代码将执行下述序列:

- 执行 `System.Object.Object()`构造函数。
- 执行 `MyBaseClass.MyBaseClass(int i)`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass(int i, int j)`构造函数。
- 执行 `MyDerivedClass.MyDerivedClass()`构造函数。

唯一的限制是使用构造函数初始化器只能指定一个构造函数。但是, 如上一个示例所示, 这并不是一个很严格的限制, 因为我们仍可以构造相当复杂的执行序列。



如果没有给构造函数指定构造函数初始化器, 编译器就会自动添加 `base()`。这会执行本节前面介绍的默认序列。

注意在定义构造函数时, 不要创建无限循环。例如:

```
public class MyBaseClass
{
    public MyBaseClass() : this(5)
    {
    }
    public MyBaseClass(int i) : this()
    {
    }
}
```

使用上述任何一个构造函数，都需要先执行另一个构造函数，而另一个构造函数需要先执行原  
告的构造函数，因此这段代码可以编译，但如果尝试实例化 `MyBaseClass`，就会得到一个  
`SystemOverflowException` 异常。

9.4 VS 和 VCE 中的 OOP 工具

OOP 在 .NET Framework 中是一个非常基础的主题，所以 VS 和 VCE 提供了几个工具来帮助开  
发 OOP 应用程序。本节就介绍其中的一些工具。

9.4.1 Class View 窗口

第 2 章介绍了 Solution Explorer 窗口与 Class View 窗口共用相同的空间。这个窗口显示了应用  
程序中的类层次结构，可供查看我们使用的类的特性。对于上一节的示例项目，其视图如图 9-3  
所示。

这个窗口分为两半，底下的一半显示了类型的成员。为了使用 Class View 窗口查看这个示例项  
目的成员和其他内容，需要显示当前隐藏的一些项。为此，应在 Class View 窗口中勾选 Class View  
Grouping 下拉列表中的项，如图 9-4 所示。

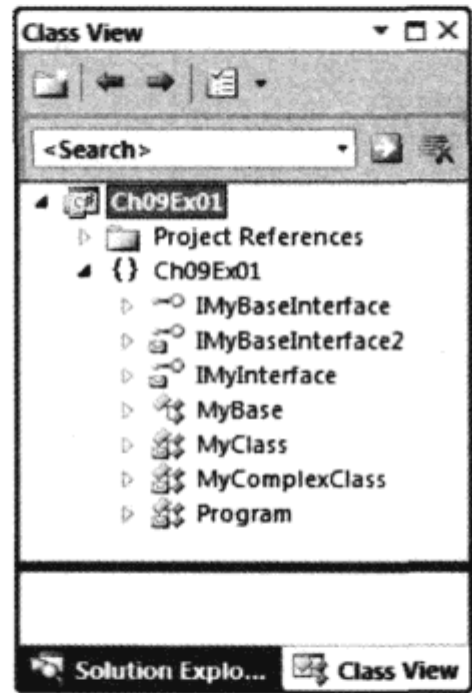


图 9-3

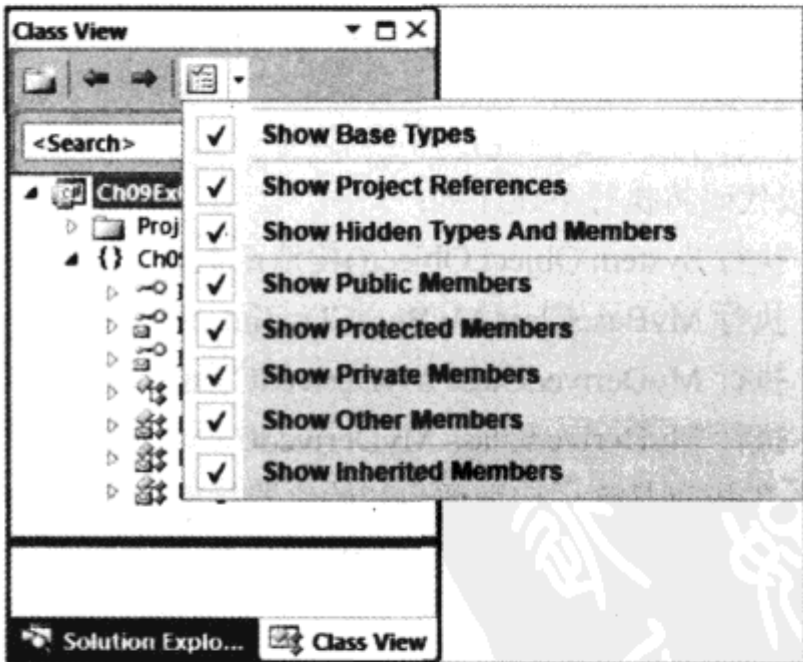


图 9-4

现在就可以看到成员和其他信息，如图 9-5 所示。

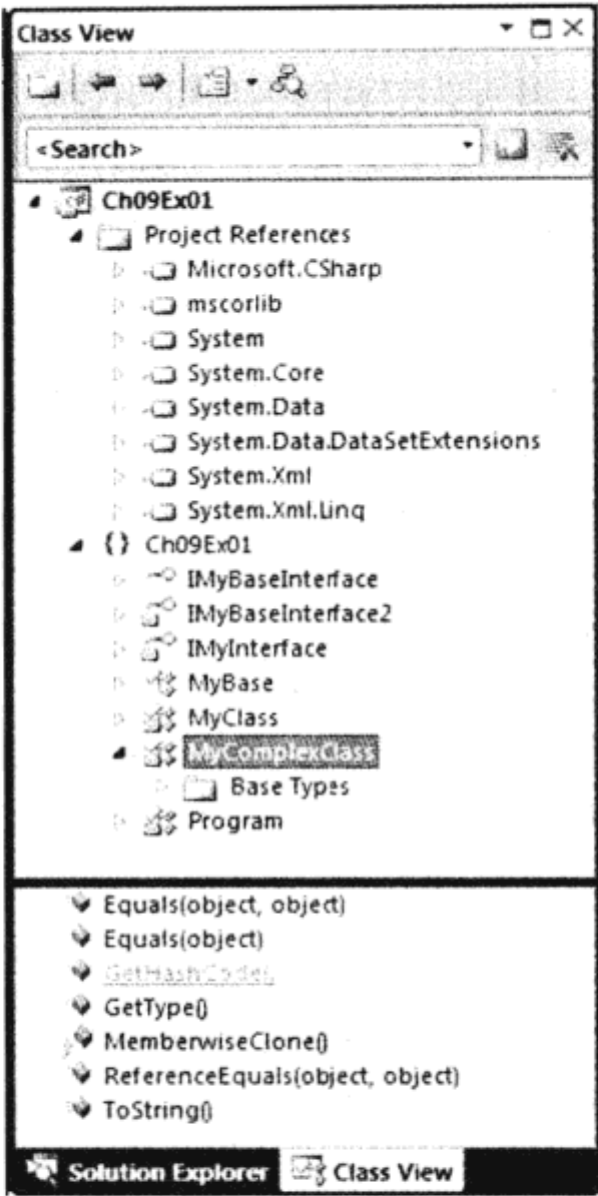


图 9-5

这里使用了许多符号，如表 9-3 中所示。

表 9-3

图 标	含 义	图 标	含 义	图 标	含 义
	项目		属性		事件
	名称空间		字段		委托
	类		结构		程序集
	接口		枚举		
	方法		枚举项		

注意，其中一些图标用于类型定义，而不是类定义，例如，枚举和结构类型。

还可以在一些项的下面放置其他符号，表示它们的访问级别(公共项没有这样的符号)，表 9-4 中列出了这些符号。

表 9-4

图 标	含 义	图 标	含 义	图 标	含 义
	私有		受保护的		内部

没有符号用于表示抽象、密封和虚拟项。

在这里除了可以查看信息外，还可以访问许多项的相关代码。双击某个项，或者右击该项，并选择 **Go To Definition**，就可以查看项目中用于定义该项的代码(假定代码是可以查看的)。如果无法查看代码，例如不能访问基类型 `System.Object` 中的代码，就应选择 **Browse Definition**，打开 **Object Browser** 视图(详见下一节)。

图 9-5 显示的另一项是 **Project References**，它可以供查看项目引用了哪些程序集，本例的项目包含 `microsoft` 和 `System` 中的核心 .NET 类型、`System.Data` 中的数据访问类型和 `System.Xml` 中的 XML 操纵类型。这里的引用也是可以扩展的，显示这些程序集中包含的名称空间和类型。

**Class View** 还可以查找代码中的类型和成员。其方法是，右击一项，选择 **Find All References**，就会在 **Find Symbol Results** 窗口中打开搜索结果列表，该窗口位于屏幕底部，是 **Error List** 显示区域的一个选项卡。还可以使用 **Class View** 给项重命名。在重命名时，可以重命名代码中出现的项的引用。也就是说，类名中不能有拼写错误，因为我们可以随时修改它们。

另外，VS2010 引入了浏览代码的一种新方式，称为调用层次结构，通过 **View Call Hierarchy** 右击菜单项就可以在 **Class View** 窗口中访问 **Call Hierarchy** 窗口。这个功能非常适于查看类成员如何彼此交互，参见下一章。

9.4.2 对象浏览器

对象浏览器(**Object Browser**)是 **Class View** 窗口的扩展版本，可以查看项目中能使用的其他类，甚至可以查看外部的类。可以自动(如上一节的情况)或手动(通过 **View | Object Browser**)进入这个窗口。这个视图显示在主窗口中，可以用与 **Class View** 窗口相同的方式浏览该视图。

这个窗口显示了与 **Class View** 窗口相同的信息，还显示了 .NET 类型的其他信息。选中某项，还可以在第三个窗口中获得该项的信息，如图 9-6 所示。

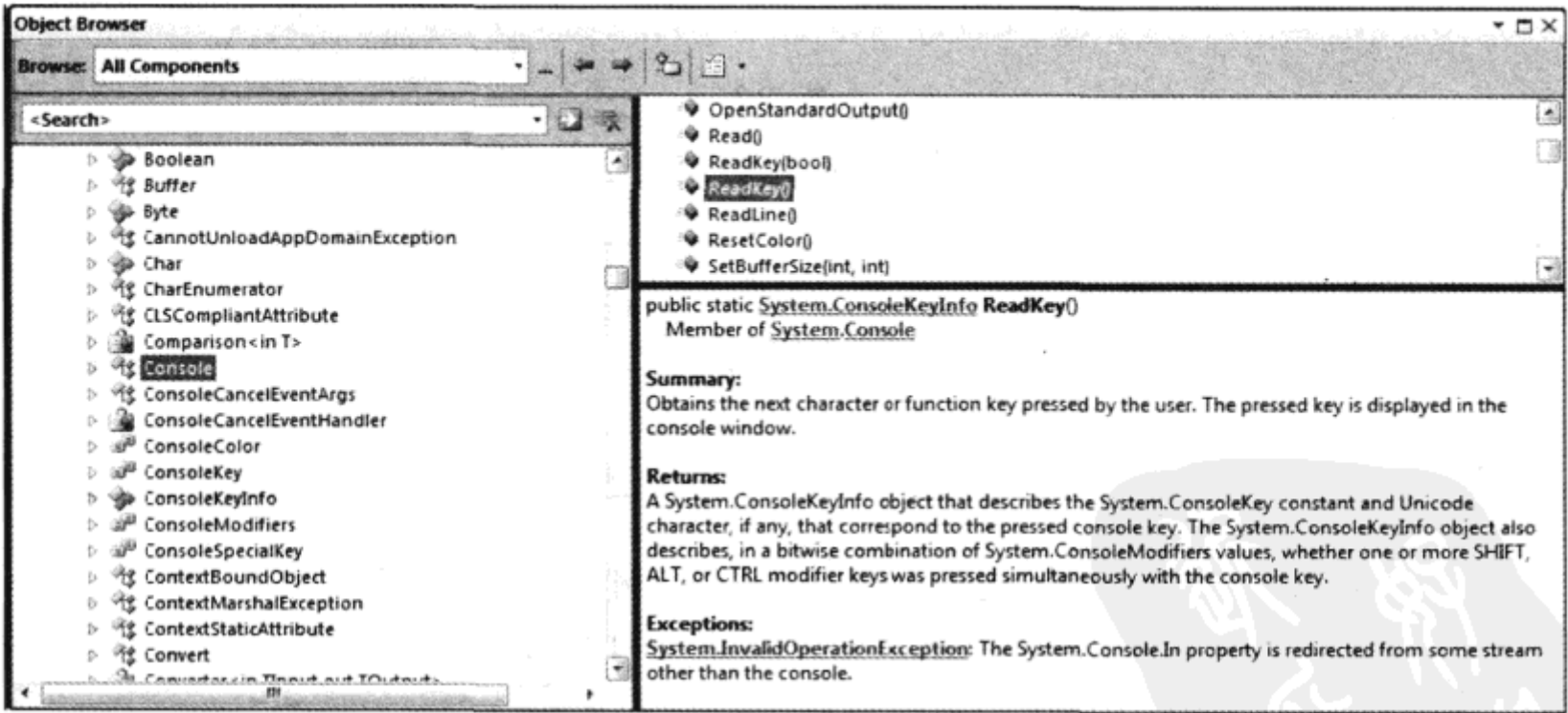


图 9-6

在图 9-6 中，选中了 `Console` 类的 `ReadKey()` 方法(`Console` 在 `microsoft` 程序集的 `System` 名称空间中)。右下角的信息窗口显示了方法签名、该方法所属的类和方法函数的小结。在研究 .NET 类型时，或者了解某个类的用途时，这些信息非常有用。



另外，还可以在自己创建的类型中使用这个信息窗口。对 Ch09Ex01 中的代码进行如下修改：



可从  
WROX.COM  
下载源代码

```
/// <summary>
/// This class contains my program!
/// </summary>
class Program
{
    static void Main(string[] args)
    {
        MyComplexClass myObj = new MyComplexClass();
        Console.WriteLine(myObj.ToString());
        Console.ReadKey();
    }
}
```

代码段 Ch09Ex01\Program.cs

然后返回到对象浏览器，就会看到这些变化反映在信息窗口中。这是 XML 文档说明的一个示例，本书不讨论 XML 文档说明，但读者有闲暇时间时，应学习这个主题。



如果手工修改上面的代码，只要键入 3 个斜杠///，IDE 就会添加输入的其他内容。它会自动分析应用于 XML 文档说明的代码，建立基本的 XML 文档说明。显然，如果需要 XML 文档说明，VS 和 VCE 就是一个很强大的工具。

### 9.4.3 添加类

VS 和 VCE 包含可以加速执行某些常见任务的工具，其中一些可以应用于 OOP。有一个 Add New Item Wizard 工具可以给项目快速添加新类，且需要键入的代码数量最少。

该工具的访问方式是单击 Project | Add New Item 菜单项，或在 Solution Explorer 窗口中右击项目，选择相应的项。采用这两种方式，都会打开一个对话框，在该对话框中，可以选择要添加的项。这个窗口的默认显示在 VS 和 VCE 中是不同的，但功能相同。在这两个 IDE 中，要添加一个类，可以在 Templates 窗口中选择 Class 项，如图 9-7 所示，为包含类的文件提供一个文件名，再单击 Add 按钮。所创建的类就以所提供的文件名命名。

在本章前面的示例中，我们在 Program.cs 文件中手动添加类定义。把类放在独立的文件中，常常可以更轻松地跟踪类。打开 Ch09Ex01 项目后，在 Add New Item 对话框中输入信息，就会在 MyNewClass.cs 中生成下列代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch09Ex01
{
```

```
class MyNewClass
{
}
}
```

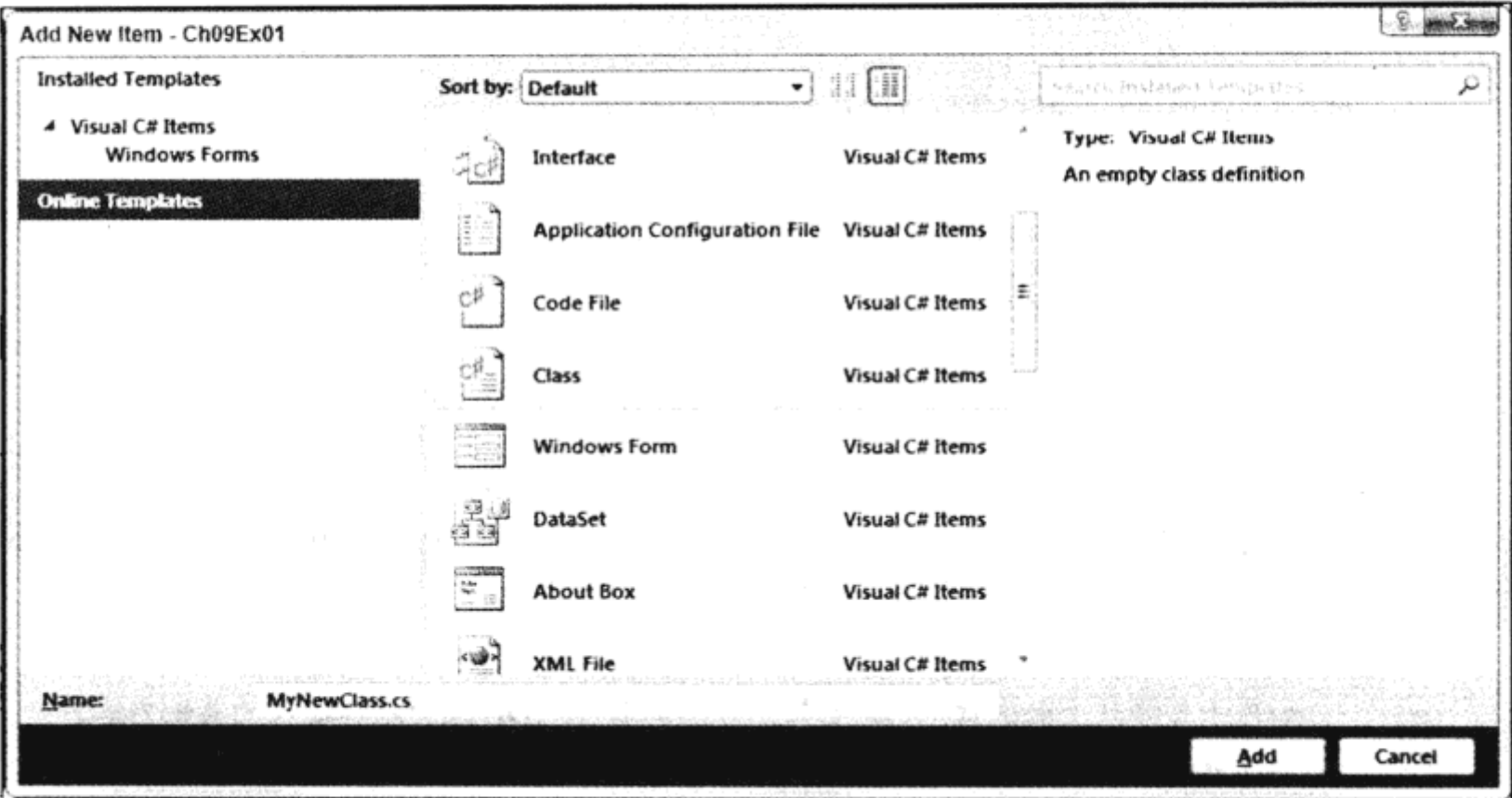



图 9-7

这个类 MyNewClass 定义在入口点类 Program 所在的名称空间中，所以可以在代码中使用它，就像它们是在相同的文件中定义一样。从代码中可以看出，生成的类不包含构造函数。如果类定义没有包含构造函数，编译器就会在编译代码时自动添加一个默认的构造函数。

9.4.4 类图

还没有介绍的 VS 的一个强大功能是从代码中生成类图，并使用类图修改项目。VS 中的类图编辑器可以很方便地为代码生成类似于 UML 的图。为了描述这个功能，下面的示例将为前面创建的 Ch09Ex01 项目生成类图。



VCE 没有类图功能，所以只能在 VS 中建立这个示例。

试一试：生成类图

- (1) 打开本章前面创建的 Ch09Ex01 项目。
- (2) 在 Solution Explorer 窗口中，选择 Program.cs，单击工具栏中的 View Class Diagram 按钮，如图 9-8 所示。

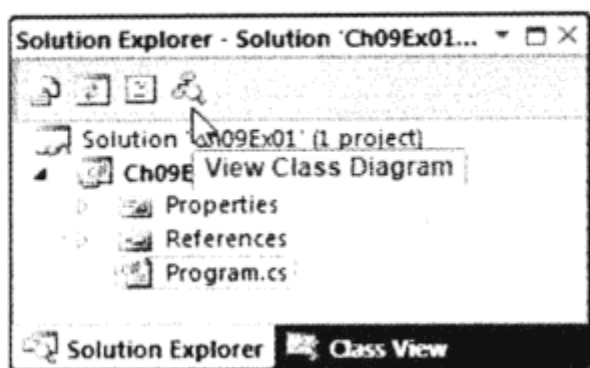


图 9-8

- (3) 显示一个类图 ClassDiagram1.cd。
- (4) 单击 IMyInterface “棒棒糖”，在 Properties 窗口中，把它的 Position 属性改为 Right。
- (5) 右击 MyBase，从上下文菜单中选择 Show Base Type 选项。
- (6) 拖动图中的对象，生成较好的布局。完成这些步骤后，类图将如图 9-9 所示。

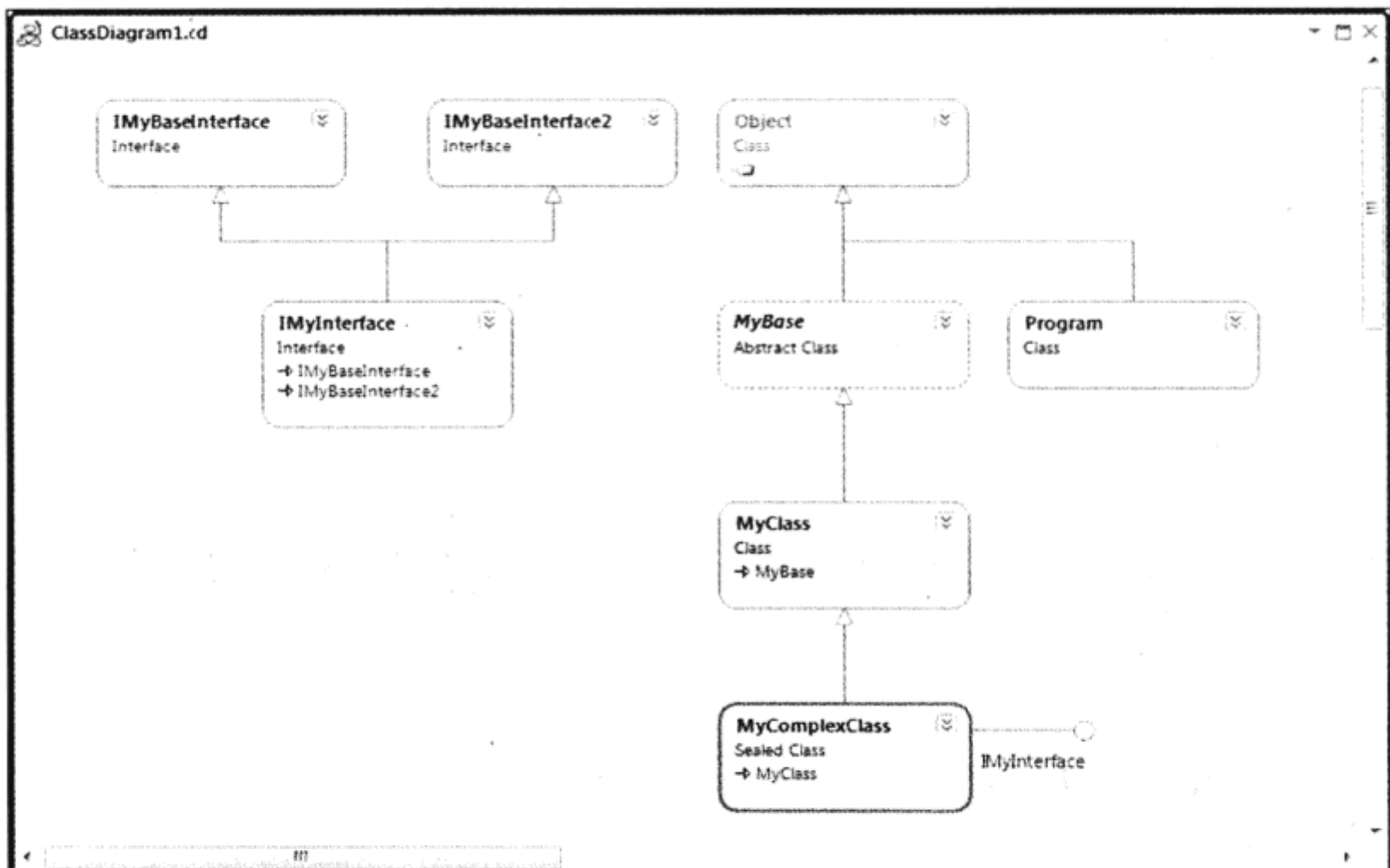


图 9-9

### 示例的说明

本例毫不费力地创建了一个与 UML 图(见图 9-2)非常类似的类图，下面的特性得到了证明：

- 类显示为蓝色框，其中包含类的名称和类型。
- 接口显示为绿色框，其中包含接口的名称和类型。
- 继承用白色箭头表示，在某些情况下，类框中包含文本。
- 实现接口的类有“棒棒糖”图标。
- 抽象类显示为虚点外框，名称显示为斜体。
- 密封类显示为粗黑外框。

单击一个对象会在屏幕底部的 Class Details 窗口中显示其他信息(如果 Class Details 窗口没有显

示出来，可以右击一个对象，选择 Class Details)。可以在此查看(和修改)类成员，还可以在 Properties 窗口中修改类的信息。



第 10 章将深入讨论如何使用类图给类添加成员。

在 Toolbox 中，可以给图添加新项，例如，类、接口和枚举等，定义图中对象之间的关系。此时，新项的代码会自动生成。

使用这个编辑器可以图形化地设计整个类型系列，而无需使用代码编辑器。显然，在实际添加功能时，必须手工完成一些工作，但这个类图编辑器是开始工作的一种绝佳方式。后面的章节还将介绍这个视图，了解它的其他用途。现在读者可以自己学习其功能。

### 9.5 类库项目

除了在项目中把类放在不同的文件中之外，还可以把它们放在完全不同的项目中。如果一个项目什么都不包含，只包含类(以及其他相关的类型定义，但没有入口点)，该项目就称为类库。

类库项目编译为.dll 程序集，在其他项目中添加对类库项目的引用，就可以访问它的内容(这可以是同一个解决方案的一部分，但这不是必须的)。这将扩展对象提供的封装性，因为类库可以进行修改和更新，而不会影响使用它们的其他项目。这意味着，您可以方便地升级类提供的服务(这会影 响多个用户应用程序)。

下面看一个类库项目的示例和一个利用该项目包含的类的独立项目。

试一试：使用类库

(1) 在 C:\BegVCSharp\Chapter09 目录中创建一个 Class Library 类型的新项目 Ch09ClassLib，如图 9-10 所示。

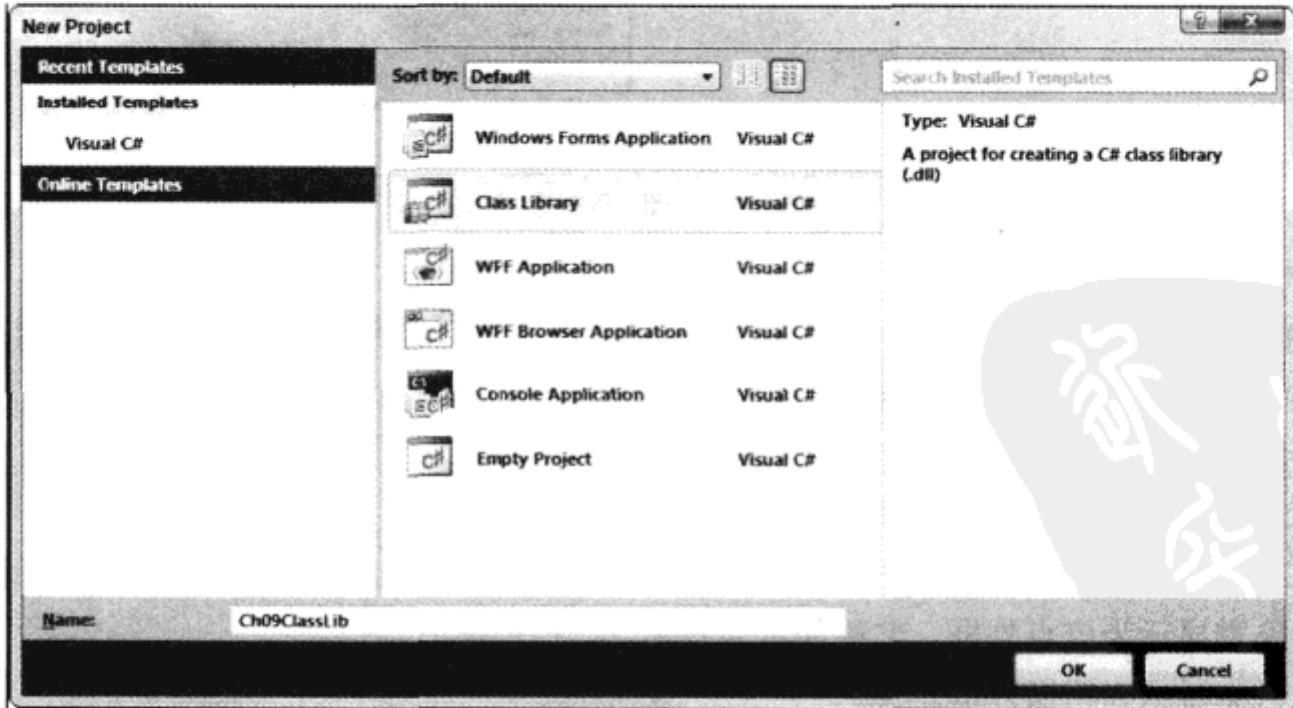


图 9-10

(2) 把文件 Class1.cs 重命名为 MyExternalClass.cs(在 Solution Explorer 窗口中右击该文件, 然后选择 Rename 来重命名该文件名)。在弹出的对话框中单击 Yes。

(3) MyExternalClass.cs 中的代码已随之自动改变, 以反映类名的改变:



可从  
WROX.COM  
下载源代码

```
public class MyExternalClass
{
}
```

代码段 Ch09ClassLib\MyExternalClass.cs

(4) 使用文件名 MyInternalClass.cs 给项目添加一个新类。

(5) 修改代码, 使类 MyInternalClass 成为内部类:



可从  
WROX.COM  
下载源代码

```
internal class MyInternalClass
{
}
```

代码段 Ch09ClassLib\MyInternalClass.cs

(6) 编译项目(注意这个项目没有入口点, 所以不能像通常那样运行它——可以选择 Build | Build Solution 菜单项来生成它)。

(7) 在 C:\BegVCSharp\Chapter09 目录中创建一个新的控制台应用程序项目 Ch09Ex02。

(8) 选择 Project | Add Reference 菜单项, 或者在 Solution Explorer 窗口中右击 References, 选择相同的选项。

(9) 单击 Browse 选项卡, 导航到 C:\BegVCSharp\Chapter09\Chapter09\Ch09ClassLib\bin\Debug\, 双击 Ch09ClassLib.dll。

(10) 完成了上述操作后, 检查引用是否已添加到 Solution Explorer 窗口中, 如图 9-11 所示。

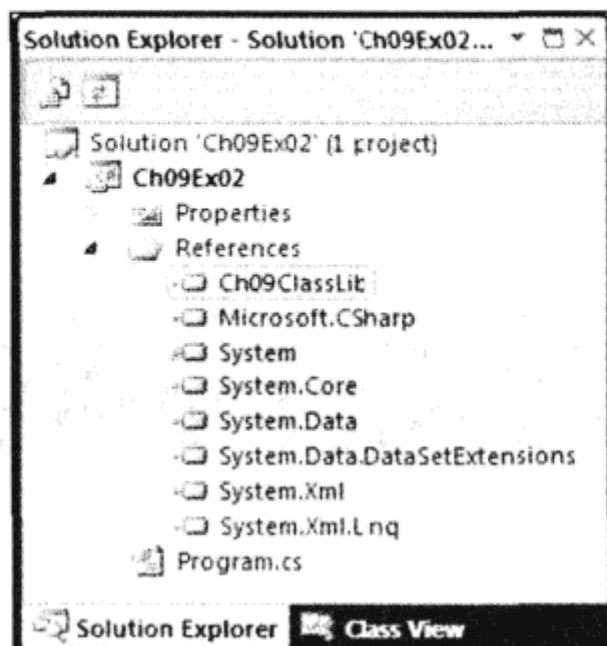


图 9-11

(11) 打开 Object Browser 窗口, 检查新引用, 看看其中包含的对象, 其结果如图 9-12 所示。

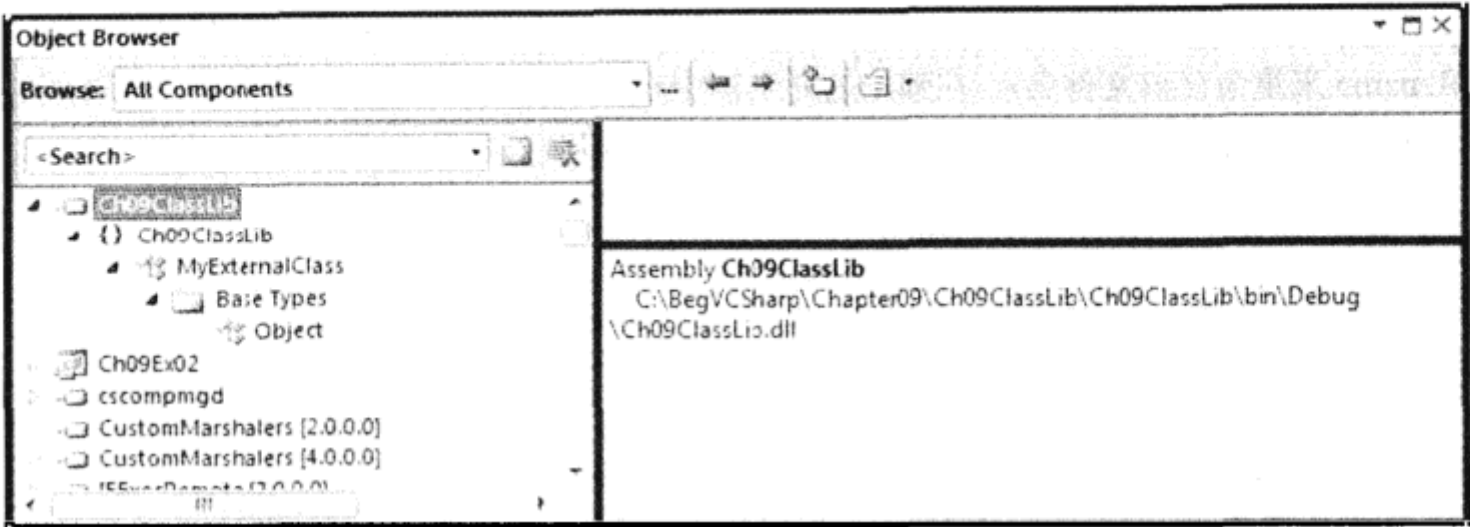


图 9-12

(12) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch09ClassLib;

namespace Ch09Ex02
{
    class Program
    {
        static void Main(string[] args)
        {
            MyExternalClass myObj = new MyExternalClass();
            Console.WriteLine(myObj.ToString());
            Console.ReadKey();
        }
    }
}
```

代码段 Ch09Ex02\Program.cs

(13) 运行应用程序，其结果如图 9-13 所示。

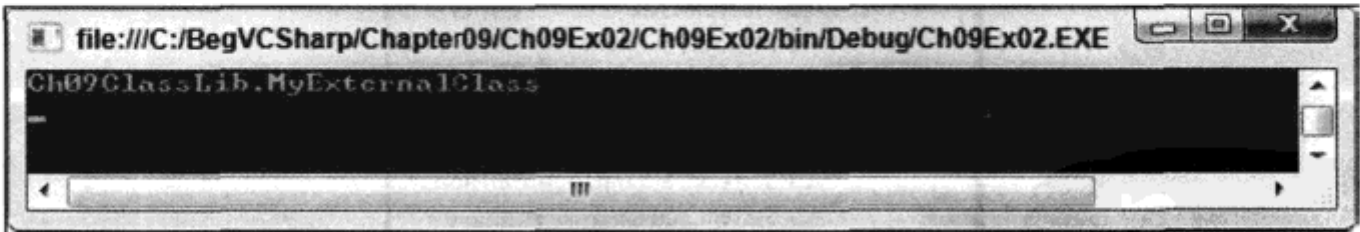


图 9-13

示例的说明

这个示例创建了两个项目，一个是类库项目，另一个是控制台应用程序项目。类库项目 Ch09ClassLib 包含两个类 MyExternalClass(可公开访问)和 MyInternalClass(只能在内部访问)。注意，默认情况下，会隐式将类确定为供内部访问，因为它没有访问修饰符。最好明确指定可访问性，因为这会使代码更容易理解，所以指令增加了 internal 关键字。控制台应用程序项目 Ch09Ex02 包



含利用类库项目的简单代码。



应用程序使用外部库中定义的类时，可以把该应用程序称为库的客户应用程序。使用所定义的类的代码一般简称为客户代码。

为了使用 Ch09ClassLib 中的类，在控制台应用程序中添加了对 Ch09ClassLib.dll 的引用。对于这个示例，该引用是指向类库的输出文件，也可以把这个文件复制到 Ch09Ex02 的本地位置上，以便继续开发类库，而不影响控制台应用程序。为了用新类库项目替换旧版本的程序集，只需用新生成的 DLL 文件覆盖旧文件即可。

在添加了引用后，就可以使用对象浏览器查看可用的类。因为类 MyInternalClass 是内部的，所以在对象浏览器窗口中看不到这个类——它不能由外部的项目访问。但是，MyExternalClass 是可供访问的，这是我们在控制台应用程序中使用的类。

可以把控制台应用程序中的代码替换为使用内部类的代码，如下所示：

```
static void Main(string[] args)
{
    MyInternalClass myObj = new MyInternalClass();
    Console.WriteLine(myObj.ToString());
    Console.ReadKey();
}
```

如果试图编译这段代码，就会产生如下编译错误：

```
'Ch09ClassLib.MyInternalClass' is inaccessible due to its protection level
```

利用外部程序集中的类的技术是使用 C#和.NET Framework 编程的关键。实际上，使用.NET Framework 中的任何类，也就是在利用外部程序集中的类，因为它们的处理方式是相同的。

## 9.6 接口和抽象类


本章介绍了如何创建接口和抽象类(现在不考虑其成员，第10章会讲述类的成员)。这两种类型在许多方面都很类似，所以应看看它们的相似和不同之处，看看哪些情况应使用什么技术。

首先讨论它们的类似之处。抽象类和接口都包含可以由派生类继承的成员。接口和抽象类都不能直接实例化，但可以声明这些类型的变量。如果这样做，就可以使用多态性把继承这两种类型的对象指定给它们的变量。接着通过这些变量来使用这些类型的成员，但不能直接访问派生对象的其他成员。

下面看看它们的区别。派生类只能继承一个基类，即只能直接继承一个抽象类(但可以用一个继承链包含多个抽象类)。相反，类可以使用任意多个接口。但这不会产生太大的区别——这两种情况取得的效果是类似的。只是采用接口的方式略有不同。

抽象类可以拥有抽象成员(没有代码体，且必须在派生类中实现，否则派生类本身必须也是抽象的)和非抽象成员(它们拥有代码体，也可以是虚拟的，这样就可以在派生类中重写)。另一方面，接口成员必须都在使用接口的类上实现——它们没有代码体。另外，按照定义，接口成员是公共的(因

为它们倾向于在外部使用), 但抽象类的成员可以是私有的(只要它们不是抽象的)、受保护的、内部的或受保护的内部成员(其中受保护的内部成员只能在应用程序的代码或派生类中访问)。此外, 接口不能包含字段、构造函数、析构函数、静态成员或常量。



抽象类主要用作对象系列的基类, 共享某些主要特性, 例如, 共同的目的和结构。接口则主要用于类, 这些类在基础水平上有所不同, 但仍可以完成某些相同的任务。

例如, 假定有一个对象系列表示火车, 基类 `Train` 包含火车的核心定义, 例如车轮的规格和引擎的类型(可以是蒸汽发动机、柴油发动机等)。但这个类是抽象的, 因为并没有“一般的”火车。为了创建一辆实际的火车, 需要给该火车添加特性。为此, 派生一些类, 例如: `PassengerTrain`、`FreightTrain` 和 `424DoubleBogey` 等, 如图 9-14 所示。

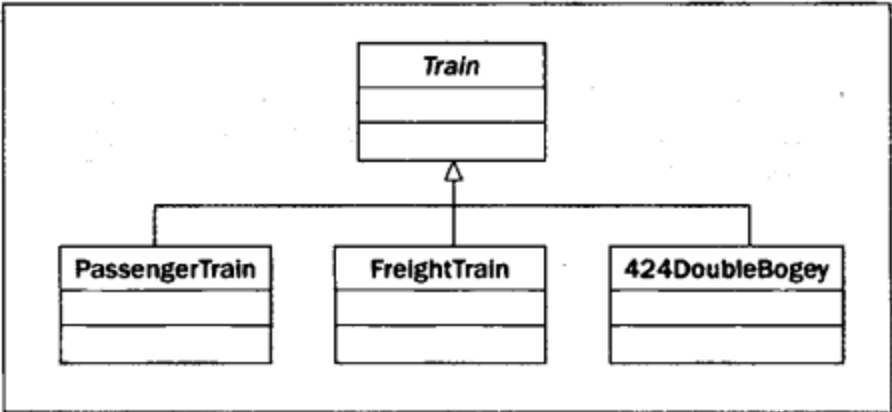


图 9-14

也可以用相同的方式来定义汽车对象系列, 使用 `Car` 抽象基类, 其派生类有 `Compact`、`SUV` 和 `PickUp`.`Car` 和 `Train` 甚至可以派生于一个相同的基类 `Vehicle`, 如图 9-15 所示。

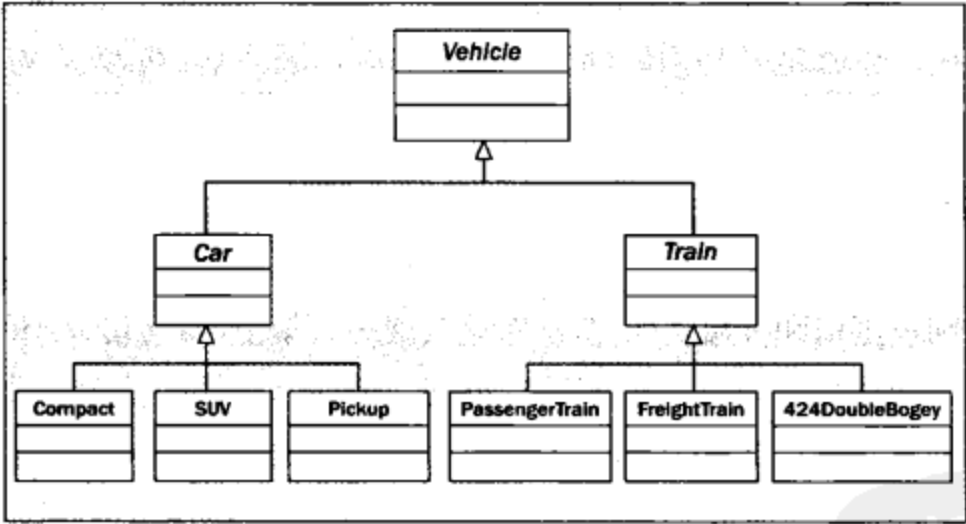


图 9-15

现在, 层次结构中的一些类共享相同的特性, 这是因为它们的目的是相同的, 而不是因为它们派生于相同的基类。例如, `PassengerTrain`、`Compact`、`SUV` 和 `Pickup` 都可以运送乘客, 所以它们都拥有 `IPassengerCarrier` 接口, `FreightTrain` 和 `Pickup` 可以运送货物, 所以它们都拥有 `IHeavyLoadCarrier` 接口, 如图 9-16 所示。

在进行更详细的细分前, 把对象系统以这种方式进行分解, 可以清晰地看到哪种情形适合使用抽象类, 哪种情形适合使用接口。只使用接口或只使用抽象继承, 就得不到这个示例的结果。

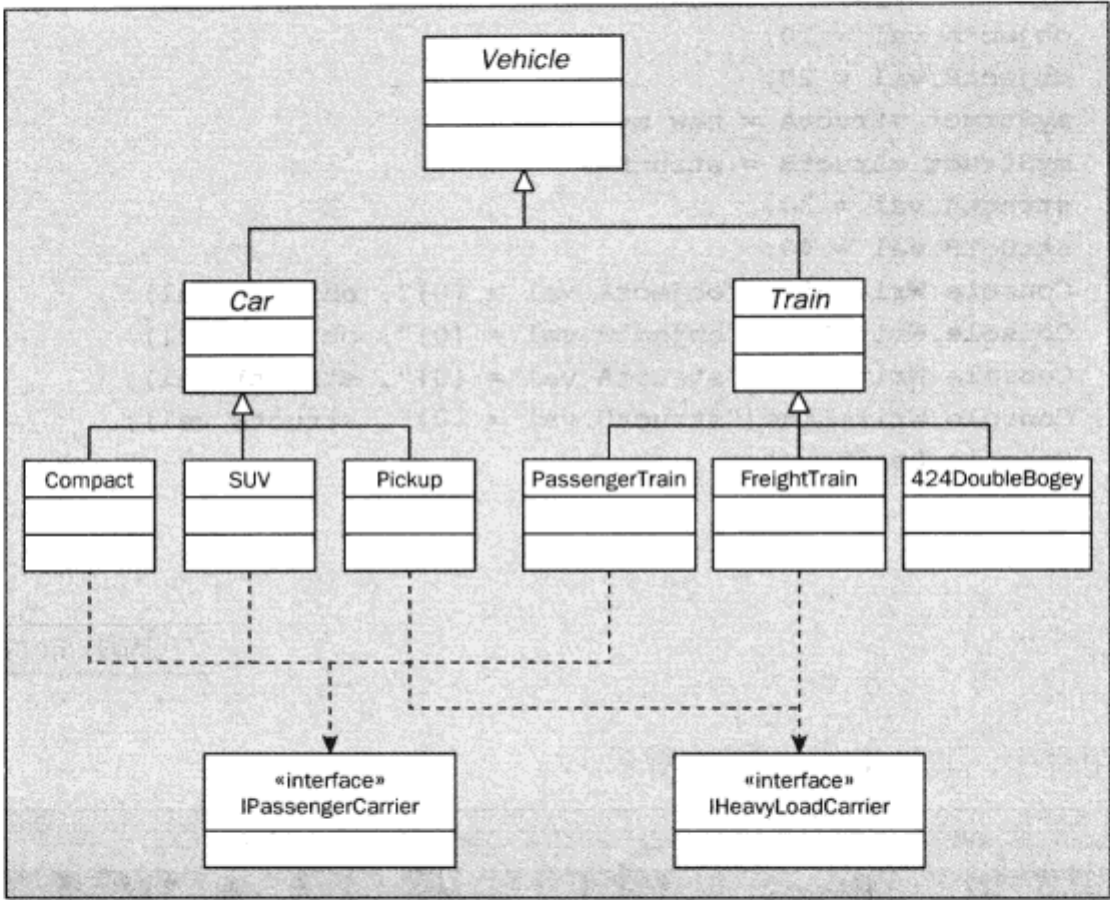


图 9-16

## 9.7 结构类型

第8章提到过结构和类非常相似，但结构是值类型，而类是引用类型。这意味着什么？最简明的方式是用一个示例来说明。

### 试一试：类和结构

- (1) 在 C:\BegVCSharp\Chapter09 目录中创建一个新控制台应用程序项目 Ch09Ex03。
- (2) 修改代码，如下所示：



可从  
wrox.com  
下载源代码

```
namespace Ch09Ex03
{
    class MyClass
    {
        public int val;
    }

    struct myStruct
    {
        public int val;
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyClass objectA = new MyClass();
        }
    }
}
```



```

        MyClass objectB = objectA;
        objectA.val = 10;
        objectB.val = 20;
        myStruct structA = new myStruct();
        myStruct structB = structA;
        structA.val = 30;
        structB.val = 40;
        Console.WriteLine("objectA.val = {0}", objectA.val);
        Console.WriteLine("objectB.val = {0}", objectB.val);
        Console.WriteLine("structA.val = {0}", structA.val);
        Console.WriteLine("structB.val = {0}", structB.val);
        Console.ReadKey();
    }
}
}

```

代码段 Ch09Ex03\Program.cs

(3) 运行应用程序，其结果如图 9-17 所示。

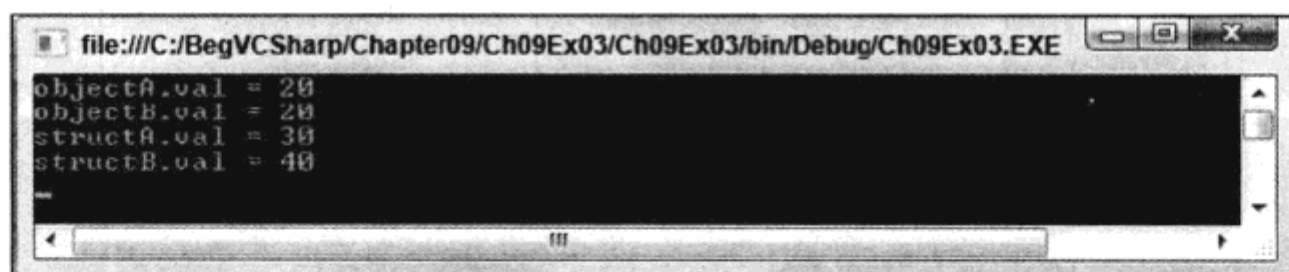


图 9-17

### 示例的说明

这个应用程序包含两个类型定义。一个是结构 `myStruct` 的定义，它有一个公共 `int` 字段 `val`，另一个是类 `MyClass` 的定义，它包含一个相同的字段(第 10 章介绍类的成员，如字段，现在只要知道它们的语法是相同的即可)。接着对这两种类型的实例执行相同的操作：

- 声明类型的变量。
- 在这个变量中创建该类型的新实例。
- 声明类型的第二个变量。
- 把第一个变量赋给第二个变量。
- 在第一个变量的实例中，给 `val` 字段赋一个值。
- 在第二个变量的实例中，给 `val` 字段赋一个值。
- 显示两个变量的 `val` 字段值。

尽管对两种类型的变量执行了相同的操作，但结果是不同的。在显示 `val` 字段的值时，两个 `object` 类型有相同的值，而结构类型有不同的值。为什么会这样？

对象是引用类型。在把对象赋给变量时，实际上是把带有一个指针的变量赋给了该指针所指向的对象。在实际代码中，指针是内存中的一个地址。在这种情况下，地址是内存中该对象所在的位置。在用下面的代码行把第一个对象引用赋给类型为 `MyClass` 的第二个变量时，实际上是复制了这个地址。

```
MyClass objectB = objectA;
```

这样两个变量就包含同一个对象的指针。

结构是值类型。其变量并不是包含结构的指针，而是包含结构本身。在用下面的代码把第一个结构赋给类型为 `myStruct` 的第二个变量时，实际上是把第一个结构的所有信息复制到另一个结构中。

```
myStruct structB = structA;
```

这个过程与本书前面介绍的简单变量类型如 `int` 是一样的。最终的结果是两个结构类型变量包含不同的结构。使用指针的全部技术隐藏在托管 C# 代码中，它使得代码更简单。使用 C# 中的不安全代码可以进行低级操作，如指针操作，但这是一个比较高级的论题，这里不予以讨论。

## 9.8 浅度和深度复制

从一个变量到另一个变量按值复制对象，而不是按引用复制对象(即以与结构相同的方式复制)可能非常复杂。因为一个对象可能包含许多其他对象的引用，例如，字段成员等，这将涉及许多烦琐的操作。把每个成员从一个对象复制到另一个对象中可能不会成功，因为其中一些成员可能是引用类型。

.NET Framework 考虑了这个问题。简单地按照成员复制对象可以通过派生于 `System.Object` 的 `MemberwiseClone()` 方法来完成，这是一个受保护的方法，但很容易在对象上定义一个调用该方法的公共方法。这个方法提供的复制功能称为浅度复制(shallow copy)，因为它没有考虑引用类型成员。因此，新对象中的引用成员就会指向与源对象中相同成员的对象，在许多情况下这并不理想。如果要创建成员的新实例(复制值，而不复制引用)，此时需要使用深度复制(deep copy)。

可以实现一个 `ICloneable` 接口，以标准的方式进行。如果使用这个接口，就必须实现它包含的 `Clone()` 方法。这个方法返回一个类型为 `System.Object` 的值。我们可以采用各种处理方式，执行所选的任何一个方法体得到这个对象。如果愿意，就可以进行深度复制(但执行过程不是必选的，所以可以按照需要执行浅度复制)。详见第 11 章。

## 9.9 小结

本章讨论了如何在 C# 中定义类和接口，把第 8 章的理论以更具体的方式表达出来。我们论述了基本声明所需要的 C# 语法和可以使用的可访问关键字，继承接口和其他类的方式，如何定义抽象和密封类以控制这种继承，以及如何定义构造函数和析构函数。

本章介绍了 `System.Object`，它是我们所定义的所有类的基类。这个类提供了几个方法，其中一些是虚拟的，所以可以重写它们的实现代码。这个类还可以把任何对象实例当作这个类的实例，对任意对象应用多态性。

我们还研究了 VS 和 VCE 为 OOP 开发提供的一些工具，包括 Class View 窗口、Object Browser 窗口，以及给项目添加新类的快速方法。在扩展“多文件”这个概念时，我们还介绍了如何创建程序集，程序集虽然不能运行，但它包含可以在其他项目中使用的类定义。

接着深入探讨了抽象类和接口，理解它们的共同和不同之处，以及使用它们的场合。

最后，讨论了引用类型和值类型，较详细地介绍了结构(对象的值类型)。这引出了浅度复制和深度复制对象的讨论，该主题将在本书的后面再次讨论。

第 10 章将介绍如何定义类成员，如属性和方法，以便在 C# 中利用 OOP 创建真正的应用程序。

9.10 练习

(1) 下面的代码存在什么错误？

```
public sealed class MyClass
{
    // Class members.
}

public class myDerivedClass : MyClass
{
    // Class members.
}
```

(2) 如何定义不能创建的类？

(3) 为什么不能创建的类(noncreatable class)仍旧有用？如何利用它们的功能？

(4) 在类库项目 Vehicles 中编写代码，实现本章前面讨论的对象系列 Vehicle，其中有 9 个对象和 2 个接口需要实现。

(5) 创建一个控制台应用程序项目 Traffic，它引用 Vehicles.dll(在第(4)题中创建)，其中包括函数 AddPassenger()，它接受任何带有 IPassengerCarrier 接口的对象。要证明代码可以运行，使用支持这个接口的每个对象实例调用该函数，在每个对象上调用派生于 System.Object 的 ToString() 方法，并把结果输出到屏幕上。

附录 A 给出了练习答案。

9.11 本章要点

主 题	重 要 概 念
类和接口定义	类用 class 关键字定义，接口用 interface 关键字定义。可以使用 public 和 internal 关键字定义类和接口的可访问性，类可以定义为 abstract 或 sealed，以控制继承性。父类和父接口在一个用逗号分隔的列表中指定，放在类或接口名和一个冒号的后面。在类定义中，只能指定一个父类，且必须是列表中的第一项
构造函数和析构函数	类自动带有默认的构造函数和析构函数的实现代码，我们很少需要提供自己的析构函数。可以使用可访问性、类名和可能需要的任何参数来定义构造函数。基类的构造函数在派生类的构造函数之前执行，使用 this 和 base 构造函数初始化器关键字，可以控制类中这些构造函数的执行顺序
类库	可以创建只包含类定义的类型库项目。这些项目不能直接执行，而必须通过客户代码在可执行程序中访问。VS 和 VCE 为创建、修改和测试类提供了各种工具
类系列	类可以组合为系列，以提供公共的操作或共享公共的特性。为此，可以从共享的基类(可以是抽象的)中继承，或者实现接口
结构定义	结构的定义方式与类非常类似，但结构是值类型，而类是引用类型
复制对象	复制对象时，必须注意应复制该对象包含的其他对象，而不是仅复制这些对象的引用。复制引用称为浅度复制，而完全复制称为深度复制。可以使用 ICloneable 接口作为一个框架，来提供类定义中的深度复制功能



# 第 10 章

## 定义类成员

### 本章内容:

---

- 如何定义类成员
- 如何使用类图添加成员
- 如何控制类成员的继承
- 如何定义嵌套的类
- 如何实现接口
- 如何使用部分类定义
- 如何使用 Call Hierarchy 窗口

本章继续讨论在 C# 中如何定义类，主要介绍的是如何定义字段、属性和方法等类成员。首先介绍每种类型需要的代码，以及如何使用向导生成相应代码的结构。我们还将论述如何通过编辑成员的属性，来快速修改这些成员。

在介绍完成员定义的基础知识后，将讨论一些比较高级的成员技术：隐藏基类成员、调用重写的基类成员、嵌套的类型定义和部分类定义。

最后将理论付诸实践，创建一个类库，以便在后面的章节中使用它。

### 10.1 成员定义

在类定义中，也提供了该类中所有成员的定义，包括字段、方法和属性。所有成员都有自己的访问级别，用下面的关键字之一来定义：

- **public**——成员可以由任何代码访问。
- **private**——成员只能由类中的代码访问(如果没有使用任何关键字，就默认使用这个关键字)。
- **internal**——成员只能由定义它的程序集(项目)内部的代码访问。
- **protected**——成员只能由类或派生类中的代码访问。

后两个关键字可以合并使用，所以也有 **protected internal** 成员。它们只能由项目(更确切地讲，

是程序集)中派生类的代码来访问。

也可以使用关键字 `static` 来声明字段、方法和属性,这表示它们是类的静态成员,而不是对象实例的成员,详见第 8 章。

### 10.1.1 定义字段

字段用标准的变量声明格式和前面介绍的修饰符来定义(可以进行初始化),例如:

```
class MyClass
{
    public int MyInt;
}
```



.NET Framework 中的公共字段以 PascalCasing 形式来命名,而不是 camelCasing。这里使用的就是这种命名方法。这就是上面的字段叫作 `MyInt` 而不是 `myInt` 的原因。这仅是推荐使用的命名模式之一,但它的意义非常重大。私有字段没有推荐的命名模式,它们通常使用 camelCasing 来命名。

字段也可以使用关键字 `readonly`,表示这个字段只能在执行构造函数的过程中赋值,或由初始化赋值语句赋值。例如:

```
class MyClass
{
    public readonly int MyInt = 17;
}
```

如本章的导言所述,字段可以使用 `static` 关键字声明为静态,例如:

```
class MyClass
{
    public static int MyInt;
}
```

静态字段必须通过定义它们的类来访问(在上面的示例中,是 `MyClass.MyInt`),而不是通过这个类的对象实例来访问。另外,可以使用关键字 `const` 来创建一个常量。按照定义, `const` 成员也是静态的,所以不需要用 `static` 修饰符(实际上,用 `static` 修饰符会产生一个错误)。

### 10.1.2 定义方法

方法使用标准函数格式、可访问性和可选的 `static` 修饰符来声明。例如:

```
class MyClass
{
    public string GetString()
    {
        return "Here is a string.";
    }
}
```



与公共字段一样, .NET Framework 中的公共方法也采用 PascalCasing 形式来命名。

注意, 如果使用了 `static` 关键字, 这个方法就只能通过类来访问, 不能通过对象实例来访问。也可以在方法定义中使用下述关键字:

- `virtual`——方法可以重写。
- `abstract`——方法必须在非抽象的派生类中重写(只用于抽象类中)。
- `override`——方法重写了一个基类方法(如果方法被重写, 就必须使用该关键字)。
- `extern`——方法定义放在其他地方。

下面的代码是方法重写的一个示例:

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        // Base implementation.
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        // Derived class implementation, overrides base implementation.
    }
}
```

如果使用了 `override`, 也可以使用 `sealed` 指定在派生类中不能对这个方法作进一步的修改, 即这个方法不能由派生类重写。例如:

```
public class MyDerivedClass : MyBaseClass
{
    public override sealed void DoSomething()
    {
        // Derived class implementation, overrides base implementation.
    }
}
```

使用 `extern` 可以在项目外部提供方法的实现代码。这是一个高级论题, 这里不做详细讨论。

### 10.1.3 定义属性

属性定义方式与字段类似, 但包含的内容比较多。如前所述, 属性涉及的内容比字段多, 是因为它们在修改状态前还可以执行一些额外的操作, 实际上, 它们可能并不修改状态。属性拥有两个类似于函数的块, 一个块用于获取属性的值, 另一个块用于设置属性的值。

这两个块也称为访问器, 分别用 `get` 和 `set` 关键字来定义, 可以用于控制对属性的访问级别。可以忽略其中的一个块来创建只读或只写属性(忽略 `get` 块创建只写属性, 忽略 `set` 块创建只读属性)。

当然，这仅适用于外部代码，因为类中的其他代码可以访问这些代码块能访问的数据。还可以在访问器上包含可访问修饰符，例如使 `get` 块变成公共的，把 `set` 块变成受保护的。只有包含其中一个一个块，才能获得有效属性(既不能读取也不能修改的属性没有任何用处)。

属性的基本结构包括标准的可访问修饰符(`public`、`private` 等)，后跟类名、属性名和 `get` 块(或 `set` 块，或者 `get` 块和 `set` 块，其中包含属性处理代码)，例如：

```
public int MyIntProp
{
    get
    {
        // Property get code.
    }
    set
    {
        // Property set code.
    }
}
```



.NET 中的公共属性也以 `PascalCasing` 方式来命名，而不是 `camelCasing` 方式命名，与字段和方法一样，这里使用 `PascalCasing` 方式。

定义代码中的第一行非常类似于定义字段的代码。区别是行末没有分号，而是一个包含嵌套 `get` 和 `set` 块的代码块。

`get` 块必须有一个属性类型的返回值，简单的属性一般与私有字段相关联，以控制对这个字段的访问，此时 `get` 块可以直接返回该字段的值，例如：

```
// Field used by property.
private int myInt;

// Property.
public int MyIntProp
{
    get
    {
        return myInt;
    }
    set
    {
        // Property set code.
    }
}
```

类外部的代码不能直接访问这个 `myInt` 字段，因为其访问级别是私有的。外部的代码必须使用属性来访问该字段。`set` 函数以类似的方式把一个值赋给字段。这里可以使用关键字 `value` 表示用户提供的属性值：

```
// Field used by property.
private int myInt;
```

```
// Property.
public int MyIntProp
{
    get
    {
        return myInt;
    }
    set
    {
        myInt = value;
    }
}
```

`value` 等于类型与属性相同的一个值，所以如果属性和字段使用相同的类型，就不必担心数据类型转换了。

这个简单的属性只能直接访问 `myInt` 字段。在对操作进行更多的控制时，属性的真正作用才能发挥出来。例如，使用下面的代码实现 `set` 块：

```
set
{
    if (value >= 0 && value <= 10)
        myInt = value;
}
```

只有赋给属性的值在 0~10 之间，才会改 `myInt`。此时，要做一个重要的设计选择：如果使用了无效值，该怎么办？有 4 种选择：

- 什么也不做(如上述代码所示)。
- 给字段赋默认值。
- 继续执行，就好像没有发生错误一样，但记录下该事件，以备将来分析。
- 抛出异常。

一般情况下，后两个选择效果较好，选择哪个选项取决于如何使用类，以及给类的用户授予多少控制权。抛出异常给用户提供的控制权相当大，可以让他们知道发生了什么情况，并作出适当的响应。为此可以使用 `System` 名称空间中的标准异常，例如：

```
set
{
    if (value >= 0 && value <= 10)
        myInt = value;
    else
        throw (new ArgumentOutOfRangeException("MyIntProp", value,
            "MyIntProp must be assigned a value between 0 and 10.));
}
```

这可以在使用属性的代码中通过 `try...catch...finally` 逻辑来处理，详见第 7 章。

记录数据，例如，记录到文本文件中，对产品代码会比较有效，因为产品代码不应发生错误。它们允许开发人员检查性能，如有必要，还可以调试现有的代码。

属性可以使用 `virtual`、`override` 和 `abstract` 关键字，就像方法一样，但这几个关键字不能用于字段。最后，如上所述，访问器可以有自己的可访问性，例如：

```
// Field used by property.
private int myInt;

// Property.
public int MyIntProp
{
    get
    {
        return myInt;
    }
    protected set
    {
        myInt = value;
    }
}
```

只有类或派生类中的代码才能使用 set 访问器。

访问器可以使用的访问修饰符取决于属性的可访问性，访问器的可访问性不能高于它所属的属性，也就是说，私有属性对它的访问器不能包含任何可访问修饰符，而公共属性可以对其访问器使用所有的可访问修饰符。下面的示例中将定义和使用字段、方法和属性。

#### 试一试：使用字段、方法和属性

- (1) 在 C:\BegVCSharp\Chapter10 目录中创建一个新控制台应用程序项目 Ch10Ex01。
- (2) 使用 Add Class 快捷方式添加一个新类 MyClass，这将在新文件 MyClass.cs 中定义这个新类。
- (3) 修改 MyClass.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
public class MyClass
{
    public readonly string Name;
    private int intVal;

    public int Val
    {
        get
        {
            return intVal;
        }
        set
        {
            if (value >= 0 && value <= 10)
                intVal = value;
            else
                throw (new ArgumentOutOfRangeException("Val", value,
                    "Val must be assigned a value between 0 and 10.));
        }
    }

    public override string ToString()
    {
        return "Name: " + Name + "\nVal: " + Val;
    }

    private MyClass() : this("Default Name")
    {

```



```

    }
    public MyClass(string newName)
    {
        Name = newName;
        intVal = 0;
    }
}

```

代码段 Ch10Ex01\MyClass.cs

(4) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

static void Main(string[] args)
{
    Console.WriteLine("Creating object myObj...");
    MyClass myObj = new MyClass("My Object");
    Console.WriteLine("myObj created.");
    for (int i = -1; i <= 0; i++)
    {
        try
        {
            Console.WriteLine("\nAttempting to assign {0} to myObj.Val...",
                               i);
            myObj.Val = i;
            Console.WriteLine("Value {0} assigned to myObj.Val.", myObj.Val);
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception {0} thrown.", e.GetType().FullName);
            Console.WriteLine("Message:\n\"{0}\"", e.Message);
        }
    }
    Console.WriteLine("\nOutputting myObj.ToString()...");
    Console.WriteLine(myObj.ToString());
    Console.WriteLine("myObj.ToString() Output.");
    Console.ReadKey();
}

```

代码段 Ch10Ex01\Program.cs

(5) 运行应用程序，其结果如图 10-1 所示。

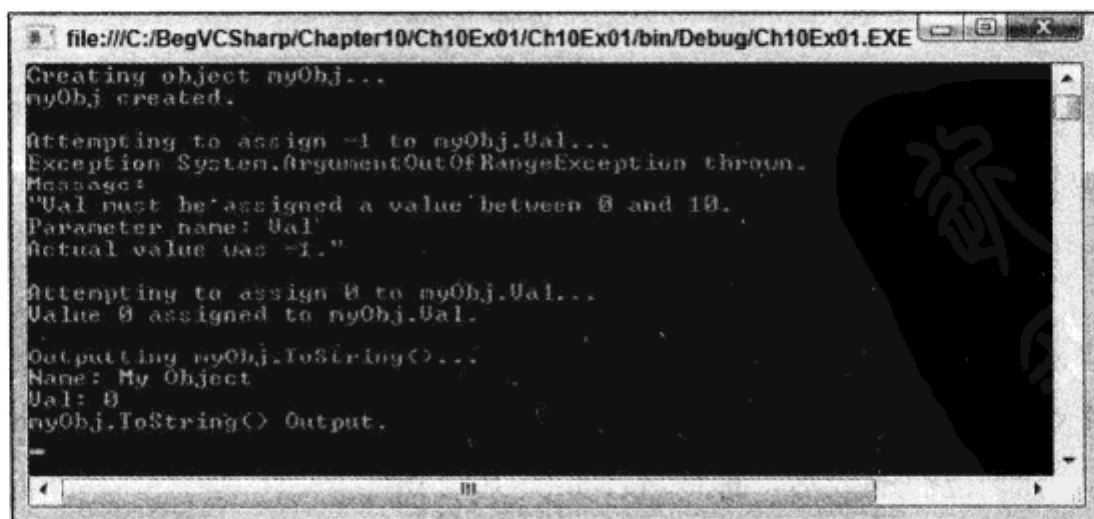


图 10-1

### 示例的说明

Main()中的代码创建并使用在 MyClass.cs 中定义的 MyClass 类的实例。实例化这个类必须使用非默认的构造函数来进行，因为 MyClass 类的默认构造函数是私有的：

```
private MyClass() : this("Default Name")
{
}
```

注意，这里用 this("Default Name")来保证，如果调用了该构造函数，Name 就获取一个值。如果这个类用于派生一个新类，这就是可能的。这是必须的，因为不给 Name 字段赋值，就会在后面产生错误。

所使用的非默认构造函数把值赋给只读字段 name(只能在字段声明或在构造函数中给它赋值)和私有字段 intVal。

接着，Main()试着给 myObj(MyClass 的实例)的 Val 属性赋值。for 循环在两次循环中赋值-1 和 0，try...catch 结构用于检查抛出的异常。把-1 赋给属性时，会抛出 System.ArgumentOutOfRangeException 类型的异常，catch 块中的代码会把该异常的信息输出到控制台窗口中。在下一个循环中，值 0 成功地赋给了 Val 属性，通过这个属性再把值赋给私有字段 intVal。

最后，使用重写的 ToString()方法输出一个格式化的字符串，来表示对象的内容：

```
public override string ToString()
{
    return "Name: " + Name + "\nVal: " + Val;
}
```

必须使用 override 关键字来声明这个方法，因为它重写了基类 System.Object 的虚拟方法 ToString()。此处的代码直接使用属性 Val，而不是私有字段 intVal，没有理由不以这种方式使用类中的属性，但这可能会对性能产生比较轻微的影响(对性能的影响非常小，我们不可能察觉到)。当然，使用属性也可以在属性中进行固有的有效性验证，这对类中的代码也是有好处的。

#### 10.1.4 在类图中添加成员

第 9 章介绍了如何使用类图研究项目中的类，还提到类图可以用于添加成员，本节就介绍这些内容。



类图功能只能在 VS 中使用，不能在 VCE 中使用。

添加和编辑成员的所有工具都显示在 Class Diagram 视图的 Class Details 窗口中。要查看这个窗口，可以为 Ch10Ex01 中的 MyClass 类创建一个类图。在类设计器中扩展类的视图(单击两个向下箭头的图标)，就可以看到已有的成员，最终视图如图 10-2 所示。

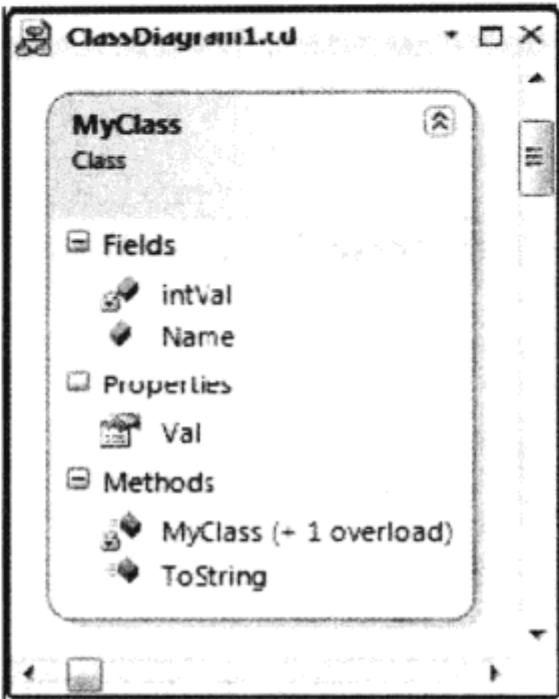


图 10-2

在 Class Details 窗口中选中类，就可以看到如图 10-3 所示的信息。

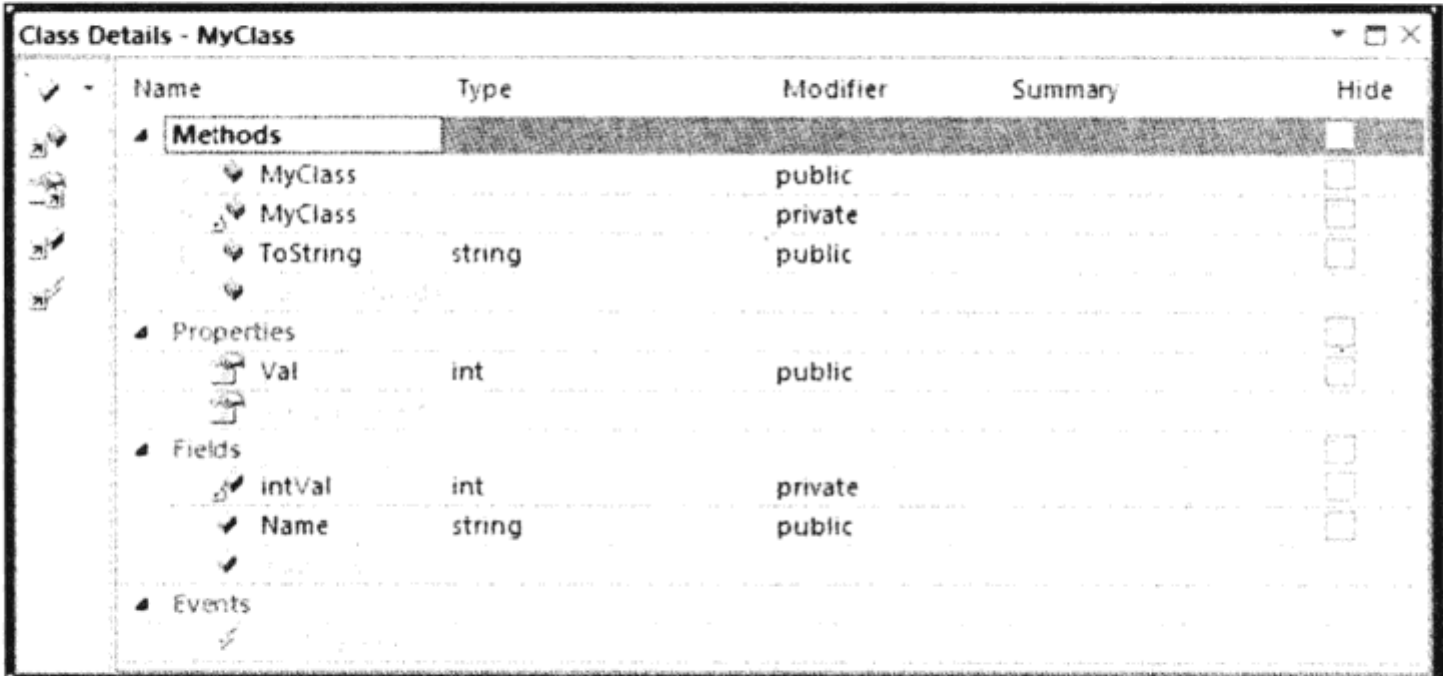


图 10-3

其中显示了当前为类定义的所有成员，并允许在相关的空间键入信息，添加新成员。

1. 添加方法

在<add method>框中键入一个方法，就可以把这个方法添加到类中。给方法命名后，就可以使用 Tab 键导航到后续的设置，从方法的返回类型开始，然后是方法的可访问性、汇总信息(它们会转换为 XML 文档说明)、是否在类图中隐藏该方法等设置。

添加好方法后，就可以按相同的方式扩展各项，添加参数。对于参数，也可以使用修饰符 out、ref 和 params。新方法的一个示例如图 10-4 所示。

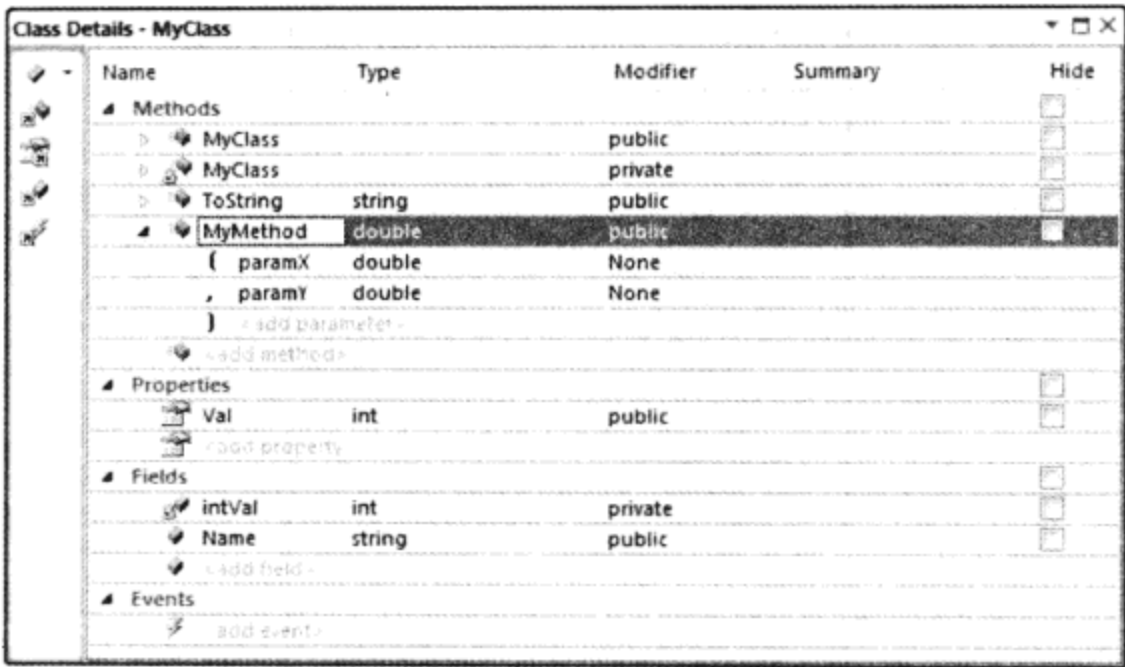


图 10-4

这个新方法在类中添加了如下代码：

```
public double MyMethod(double paramX, double ParamY)
{
    Throw new System.NotImplementedException();
}
```

方法的其他配置可以在 **Properties** 窗口中完成，如图 10-5 所示。

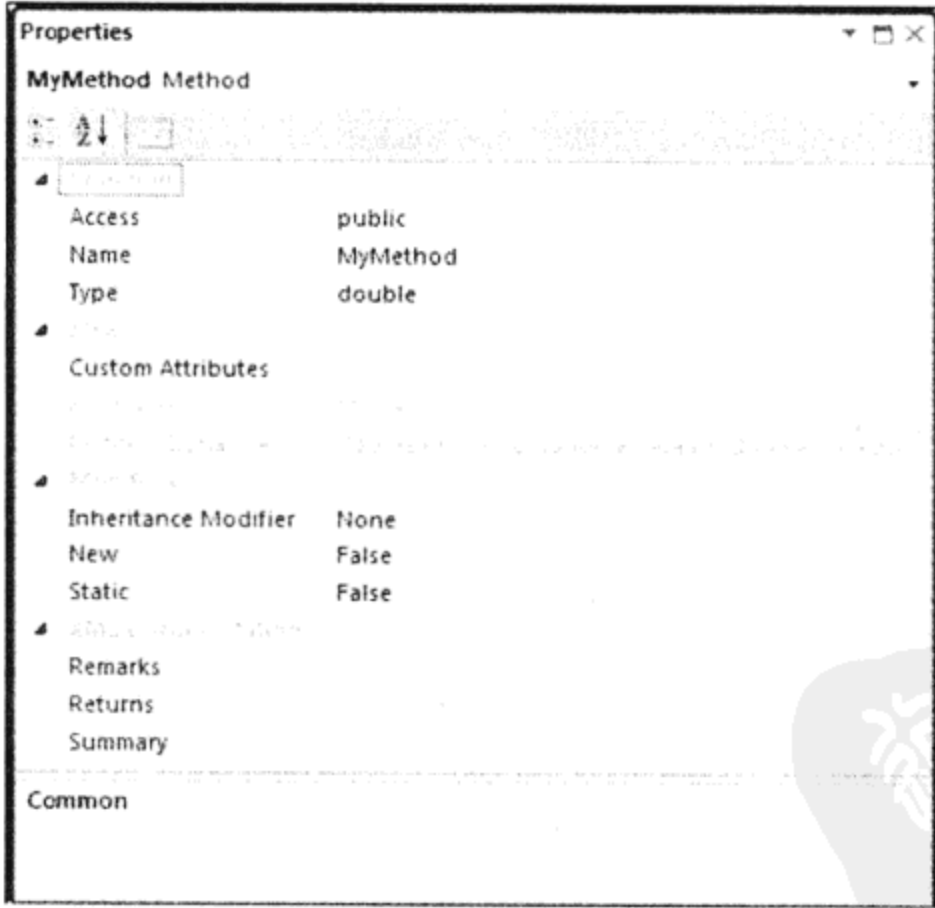


图 10-5

在这个窗口中可以把方法设置为静态的。显然，这种技术不能提供方法的实现代码，但提供了基本结构，肯定可以减少键入错误！

2. 添加属性

可以采用相同的方式添加属性。图 10-6 显示了使用 Class Details 窗口添加的新属性。

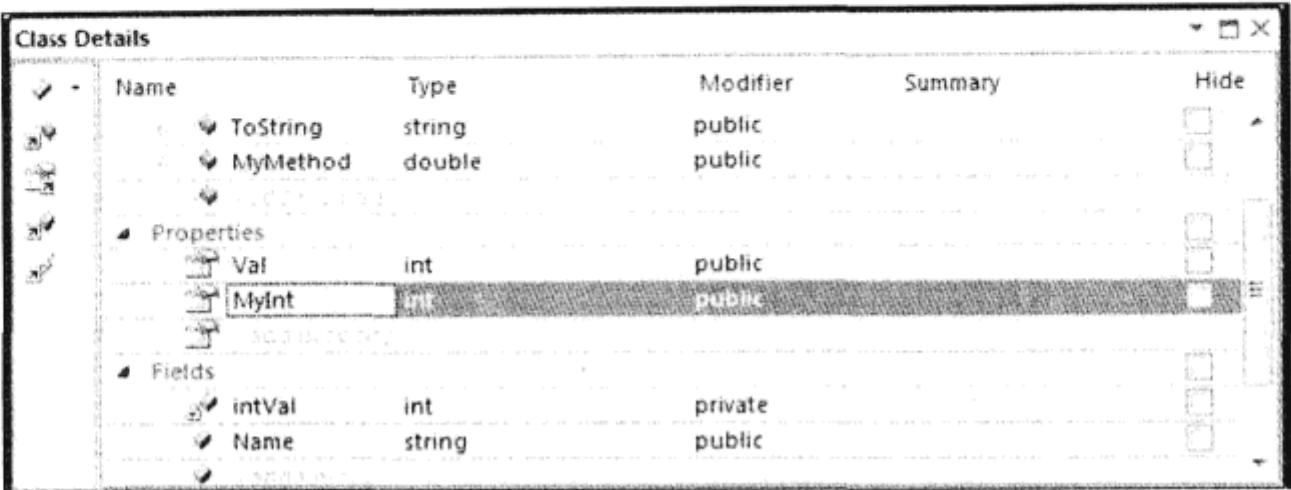


图 10-6

这会添加如下属性：

```
public int myInt
{
    get
    {
        throw new System.NotImplementedException();
    }
    set {}
}
```

注意，该窗口没有提供完整的实现代码，您需要自己去完成，包括为简单的属性匹配一个带字段的属性，删除访问器(把属性设置为只读或只写的)，或者给访问器应用可访问修饰符。该窗口提供了基本结构。

3. 添加字段

添加字段是很简单的。只需键入字段的名称，选择类型和访问修饰符即可。

10.1.5 重构成员

在添加属性时有一项很方便的技术，可以从字段中生成属性，下面是一个重构(refactoring)的示例，“重构”表示使用工具修改代码，而不是手工修改。为此，只需右击类图中的某个成员，或者在代码视图中右击某个成员即可。

VCE 包含有限的重构功能，但不包含这里介绍的字段封装功能。在这方面，VS 提供的选项要远远多于VCE。

例如，如果 MyClass 类包含如下字段：

```
public string myString;
```

右击该字段，选择 Refactor | Encapsulate Field，就会打开如图 10-7 所示的对话框。

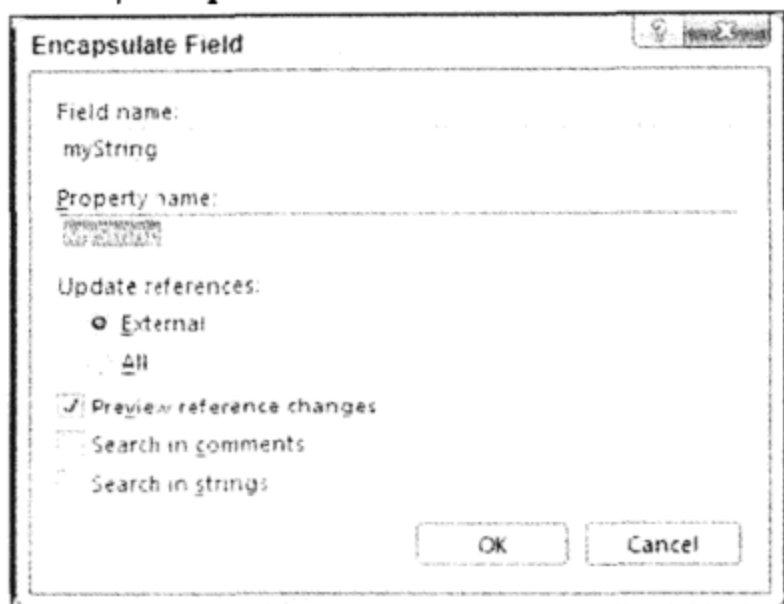


图 10-7

接受默认选项，就会修改 MyClass 的代码，如下所示：

```
Private string myString;
public string MyString;
{
    get
    {
        Return myString;
    }
    set
    {
        myString = value;
    }
}
```

myString 字段的可访问性变成了 private，同时创建了一个公共属性 MyString，它自动链接到 myString 上。这会减少单纯为字段创建属性的时间。

### 10.1.6 自动属性

属性是访问对象状态的首选方式，因为它们禁止外部代码实现对象内部的数据存储机制。属性还对内部数据的访问方式施加了更多的控制，本章代码在多处体现了这一点。但是，一般以非常标准的方式定义属性，即通过一个公共属性来直接访问一个私有成员。其代码非常类似于上一节的代码，这是 VS 重构工具自动生成的。

重构功能肯定加快了键入速度，C#还为此提供了另一种方式：自动属性。利用自动属性，可以用简化的语法声明属性，C#编译器会自动添加未键入的内容。具体而言，编译器会声明一个用于存储属性的私有字段，并在属性的 get 和 set 块中使用该字段，我们无需考虑细节。

使用下面的代码结构就可以定义一个自动属性：

```
public int MyIntProp
{
    get;
    set;
}
```



甚至可以在一行代码上定义自动属性，以便节省空间，而不会过度地降低属性的可读性：

```
public int MyIntProp { get; set; }
```

我们按照通常的方式定义属性的可访问性、类型和名称，但没有给 `get` 和 `set` 块提供实现代码。这些块的实现代码(和底层的字段)都由编译器提供。

使用自动属性时，只能通过属性访问数据，不能通过底层的私有字段来访问，因为我们不知道底层私有字段的名称(该名称是在编译期间定义的)。但这并不是一个真正意义上的限制，因为可以直接使用属性名。自动属性的唯一限制是它们必须包含 `get` 和 `set` 存取器，无法使用这种方式定义只读或只写属性。

## 10.2 类成员的其他议题

下面该讨论一些比较高级的成员议题了。本节主要研究：

- 隐藏基类方法
- 调用重写或隐藏的基类方法
- 嵌套的类型定义

### 10.2.1 隐藏基类方法

当从基类继承一个(非抽象的)成员时，也就继承了其实现代码。如果继承的成员是虚拟的，就可以用 `override` 关键字重写这段实现代码。无论继承的成员是否为虚拟，都可以隐藏这些实现代码。这是很有用的，例如，当继承的公共成员不像预期的那样工作时，就可以隐藏它。

使用下面的代码就可以隐藏：

```
public class MyBaseClass
{
    public void DoSomething()
    {
        // Base implementation.
    }
}

public class MyDerivedClass : MyBaseClass
{
    public void DoSomething()
    {
        // Derived class implementation, hides base implementation.
    }
}
```

尽管这段代码正常运行，但它会产生一个警告，说明隐藏了一个基类成员。如果是无意间隐藏了一个需要使用的成员，此时就可以改正错误。如果确实要隐藏该成员，就可以使用 `new` 关键字显式地表明意图：

```
public class MyDerivedClass : MyBaseClass
{

```

```
new public void DoSomething()
{
    // Derived class implementation, hides base implementation.
}
}
```

其工作方式是完全相同的，但不会显示警告。此时应注意隐藏基类成员和重写它们的区别。考虑下面的代码：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

其中重写方法将替换基类中的实现代码，这样，下面的代码就将使用新版本，即使这是通过基类类型进行的，情况也是这样(使用多态性)：

```
MyDerivedClass myObj = new MyDerivedClass();
MyBaseClass myBaseObj;
myBaseObj = myObj;
myBaseObj.DoSomething();
```

结果如下：

Derived imp

另外，还可以使用下面的代码隐藏基类方法：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}

public class MyDerivedClass : MyBaseClass
{
    new public void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

基类方法不必是虚拟的，但结果是一样的，只需修改上面代码中的一行即可。对于基类的虚拟方法和非虚拟方法来说，其结果如下：

```
Base imp
```

尽管隐藏了基类的实现代码，但仍可以通过基类访问它。

### 10.2.2 调用重写或隐藏的基类方法

无论是重写成员还是隐藏成员，都可以在派生类的内部访问基类成员。这在许多情况下都是有用的，例如：

- 要对派生类的用户隐藏继承的公共成员，但仍能在类中访问其功能。
- 要给继承的虚拟成员添加实现代码，而不是简单地用重写的新执行代码替换它。

为此，可以使用 `base` 关键字，它表示包含在派生类中的基类的实现代码(在控制构造函数时，其用法是类似的，如第 9 章所述)，例如：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        // Base implementation.
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        // Derived class implementation, extends base class implementation.
        base.DoSomething();
        // More derived class implementation.
    }
}
```

这段代码执行包含在 `MyBaseClass` 中的 `DoSomething()` 版本，`MyBaseClass` 是 `MyDerivedClass` 的基类，而 `DoSomething()` 版本包含在 `MyDerivedClass` 中。因为 `base` 使用的是对象实例，所以在静态成员中使用它会产生错误。

#### this 关键字

除了使用第 9 章的 `base` 关键字外，还可以使用 `this` 关键字。与 `base` 一样，`this` 也可以用在类成员的内部，且该关键字也引用对象实例。只是 `this` 引用的是当前的对象实例(即不能在静态成员中使用 `this` 关键字，因为静态成员不是对象实例的一部分)。

`this` 关键字最常用的功能是把当前对象实例的引用传递给一个方法，如下例所示：

```
public void doSomething()
{
    MyTargetClass myObj = new MyTargetClass();
    myObj.DoSomethingWith(this);
}
```

其中，被实例化的 `MyTargetClass` 实例有一个 `DoSomethingWith()` 方法，该方法带一个参数，其

类型与包含上述方法的类兼容。这个参数类型可以是类的类型、由这个类继承的类类型，或者由这个类或 `System.Object` 实现的一个接口。

`this` 关键字的另一个常见用法是限定本地类型的成员，例如：

```
public class MyClass
{
    private int someData;

    public int SomeData
    {
        get
        {
            return this.someData;
        }
    }
}
```

许多开发人员都喜欢这个语法，它可以用于任意成员类型，因为可以一眼看出引用的是成员，而不是局部变量。

### 10.2.3 嵌套的类型定义

除了在名称空间中定义类型之外，还可以在其他类中定义这些类。如果这么做，就可以在定义中使用各种访问修饰符，而不仅仅是 `public` 和 `internal`，也可以使用 `new` 关键字隐藏继承于基类的类型定义。例如，下面的代码定义了 `MyClass`，也定义了一个嵌套的类 `myNestedClass`：

```
public class MyClass
{
    public class myNestedClass
    {
        public int nestedClassField;
    }
}
```

如果要在 `MyClass` 的外部实例化 `myNestedClass`，就必须限定名称，例如：

```
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

但是，如果嵌套的类声明为私有，或者声明为其他与执行该实例化的代码不兼容的访问级别，就不能这么做。这个功能主要用于定义对于其包含类来说是私有的类，这样，名称空间中的其他代码就不能访问它。

## 10.3 接口的实现

在继续前，先讨论一下如何定义和实现接口。第 9 章介绍了接口定义的方式与类相似，使用的代码如下：

```
interface IMyInterface
{
    // Interface members.
}
```

接口成员的定义与类成员的定义相似，但有几个重要的区别：

- 不允许使用访问修饰符(public、private、protected 或 internal)，所有的接口成员都是公共的。
- 接口成员不能包含代码体。
- 接口不能定义字段成员。
- 接口成员不能用关键字 static、virtual、abstract 或 sealed 来定义。
- 类型定义成员是禁止的。

但要隐藏继承了基接口的成员，可以用关键字 new 来定义它们，例如：

```
interface IMyBaseInterface
{
    void DoSomething();
}

interface IMyDerivedInterface : IMyBaseInterface
{
    new void DoSomething();
}
```

其执行方式与隐藏继承的类成员的方式一样。

在接口中定义的属性可以定义访问块 get 和 set 中的哪一个能用于该属性(或将它们同时用于该属性)，例如：

```
interface IMyInterface
{
    int MyInt { get; set; }
}
```

其中 int 属性 MyInt 有 get 和 set 存取器。对于访问级别有更严限制的属性来说，可以省略它们中的任一个。



这个语法类似于自动属性，但自动属性是为类(而不是接口)定义的，自动属性必须包含 get 和 set 存取器。

接口没有指定应如何存储属性数据。接口不能指定字段，例如用于存储属性数据的字段。最后，接口与类一样，可以定义为类的成员(但不能定义为其他接口的成员，因为接口不能包含类型定义)。

## 在类中实现接口

实现接口的类必须包含该接口所有成员的实现代码，且必须匹配指定的签名(包括匹配指定的 get 和 set 块)，并且必须是公共的。例如：

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}
```

```
public class MyClass : IMyInterface
{
    public void DoSomething()
    {
    }

    public void DoSomethingElse()
    {
    }
}
```

可以使用关键字 `virtual` 或 `abstract` 来实现接口成员，但不能使用 `static` 或 `const`。还可以在基类上实现接口成员，例如：

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}

public class MyBaseClass
{
    public void DoSomething()
    {
    }
}

public class MyDerivedClass : MyBaseClass, IMyInterface
{
    public void DoSomethingElse()
    {
    }
}
```

继承一个实现给定接口的基类，就意味着派生类隐式地支持这个接口，例如：

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}

public class MyBaseClass : IMyInterface
{
    public virtual void DoSomething()
    {
    }

    public virtual void DoSomethingElse()
    {
    }
}

public class MyDerivedClass : MyBaseClass
```



```

{
    public override void DoSomething()
    {
    }
}

```

显然，在基类中把实现代码定义为虚拟，派生类就可以替换该实现代码，而不是隐藏它们。如果要使用 `new` 关键字隐藏一个基类成员，而不是重写它，则方法 `IMyInterface.DoSomething()` 就总是引用基类版本，即使通过这个接口来访问派生类，也是这样。

### 1. 显式实现接口成员

也可以由类显式地实现接口成员。如果这么做，该成员就只能通过接口来访问，不能通过类来访问。上一节的代码中使用的隐式成员可以通过类和接口来访问。

例如，如果类 `MyClass` 隐式地实现接口 `IMyInterface` 的方法 `DoSomething()`，如上所述，则下面的代码就是有效的：

```

MyClass myObj = new MyClass();
myObj.DoSomething();

```

下面的代码也是有效的：

```

MyClass myObj = new MyClass();
IMyInterface myInt = myObj;
myInt.DoSomething();

```

另外，如果 `MyDerivedClass` 显式实现 `DoSomething()`，就只能使用后一种技术。其代码如下：

```

public class MyClass : IMyInterface
{
    void IMyInterface.DoSomething()
    {
    }

    public void DoSomethingElse()
    {
    }
}

```

其中 `DoSomething()` 是显式实现的，而 `DoSomethingElse()` 是隐式实现的。只有后者可以直接通过 `MyClass` 的对象实例来访问。

### 2. 用非公共的可访问性添加属性存取器

前面说过，如果实现带属性的接口，就必须实现匹配的 `get/set` 存取器。这并不是绝对正确的——如果在定义属性的接口中只包含 `set` 块，就可给类中的属性添加 `get` 块，反之亦然。但是，只有所添加的存取器的可访问修饰符比接口中定义的存取器的可访问修饰符更严格时，才能这么做。因为按照定义，接口定义的存取器是公共的，也就是说，只能添加非公共的存取器。例如：

```

public interface IMyInterface
{

```

```
    int MyIntProperty
    {
        get;
    }
}

public class MyBaseClass : IMyInterface
{
    public int MyIntProperty { get; protected set; }
}
```

## 10.4 部分类定义

如果所创建的类包含一种类型或其他类型的许多成员时，就很容易混淆，代码文件也比较长。这里可以采用前面章节介绍的一种方法，即给代码分组。在代码中定义区域，就可以折叠和展开各个代码区，使代码更便于阅读。例如，有一个类的定义如下：

```
public class MyClass
{
    #region Fields
    private int myInt;
    #endregion

    #region Constructor
    public MyClass()
    {
        myInt = 99;
    }
    #endregion

    #region Properties
    public int MyInt
    {
        get
        {
            return myInt;
        }

        set
        {
            myInt = value;
        }
    }
    #endregion

    #region Methods
    public void DoSomething()
    {
        // Do something...
    }
    #endregion
}
```

上述代码可以展开和折叠类的字段、属性、构造函数和方法，以便集中精力考虑自己感兴趣的内容。甚至可以按这种方式嵌套各个区域，这样一些区域就只能在包含它们的区域被展开后才能看到。

但是，即便使用这种技术，代码也可能难以理解。对此，一种方法是使用部分类定义(**partial class definition**)。简言之，就是使用部分类定义，把类的定义放在多个文件中。例如，可以把字段、属性和构造函数放在一个文件中，而把方法放在另一个文件中。为此，只需在每个包含部分类定义的文件中对类使用 **partial** 关键字即可，如下所示：

```
public partial class MyClass
{
    ...
}
```

如果使用部分类定义，**partial** 关键字就必须出现在包含定义部分的每个文件的与此相同的位置。

部分类对 Windows 应用程序隐藏与窗体布局相关的代码有很大的作用。第 2 章已经介绍了这些内容。在 `Form1` 类中，Windows 窗体的代码存储在 `Form1.cs` 和 `Form1.Designer.cs` 中，这样就可以主要考虑窗体的功能，无需担心代码会被自己不感兴趣的信息搅乱。

对于部分类，最后要注意的一点是：应用于部分类的接口也会应用于整个类，也就是说，下面的两个定义：

```
public partial class MyClass : IMyInteface1
{
    ...
}
```

```
public partial class MyClass : IMyInteface2
{
    ...
}
```

和

```
public class MyClass : IMyInteface1, IMyInteface2
{
    ...
}
```

是等价的。

部分类定义可以在一个部分类定义文件或者多个部分类定义文件中包含基类。但如果基类在多个定义文件中指定，它就必须是同一个基类，因为在 C# 中，类只能继承一个基类。

## 10.5 部分方法定义

部分类也可以定义部分方法。部分方法在部分类中定义，但没有方法体，在另一个部分类中包含实现代码。在这两个部分类中，都要使用 **partial** 关键字。

```
public partial class MyClass
{
    partial void MyPartialMethod();
}
```

```
public partial class MyClass
{
    partial void MyPartialMethod()
    {
        // Method implementation
    }
}
```

部分方法也可以是静态的，但它们总是私有的，且不能有返回值。它们使用的任何参数都不能是out参数，但可以是ref参数。部分方法也不能使用 virtual、abstract、override、new、sealed 和 extern 修饰符。

有了这些限制，就不太容易看出部分方法的作用了。实际上，部分方法在编译代码时非常重要，其用法倒并不重要。考虑下面的代码：

```
public partial class MyClass
{
    partial void DoSomethingElse();

    public void DoSomething()
    {
        Console.WriteLine("DoSomething() execution started.");
        DoSomethingElse();
        Console.WriteLine("DoSomething() execution finished.");
    }
}

public partial class MyClass
{
    partial void DoSomethingElse()
    {
        Console.WriteLine("DoSomethingElse() called.");
    }
}
```

在第一个部分类定义中定义和调用部分方法DoSomethingElse，在第二个部分类中实现它。在控制台应用程序中调用DoSomething时，输出如下内容：

```
DoSomething() execution started.
DoSomethingElse() called.
DoSomething() execution finished.
```

如果删除第二个部分类定义，或者删除部分方法的全部执行代码(注释掉代码)，输出就如下所示：

```
DoSomething() execution started.
DoSomething() execution finished.
```

读者可能认为，调用 `DoSomethingElse` 时，运行库发现该方法没有实现代码，因此会继续执行下一行代码。但实际上，编译代码时，如果代码包含一个没有实现代码的部分方法，编译器会完全删除该方法，还会删除对该方法的所有调用。执行代码时，不会检查实现代码，因为没有检查方法的调用。这会略微提高性能。

与部分类一样，在定制自动生成的代码或设计器创建的代码时，部分方法是很有用的。设计器会声明部分方法，用户根据具体情形选择是否实现它。如果不实现它，就不会影响性能，因为该方法在编译过的代码中不存在。

现在考虑为什么部分方法不能有返回类型。如果可以回答这个问题，就可以确保完全理解了这个问题，我们将此留作练习。

## 10.6 示例应用程序

为了解释前面使用的一些技术，下面开发一个类模块，以便在后续章节中使用。这个类模块包含两个类：

- **Card**——表示一张标准的扑克牌，包含梅花、方块、红心和黑桃，其顺序是从 A 到 K。
- **Deck**——表示一副完整的 52 张扑克牌，在扑克牌中可以按照位置访问各张牌，并可以洗牌。

再开发一个简单的客户程序，确保程序正常工作，但在整个扑克牌应用程序中不使用扑克牌。

### 10.6.1 规划应用程序

这个应用程序的类库 `Ch10CardLib` 包含类。在开始编写代码前，应规划一下需要的结构和类的功能。

#### 1. Card 类

`Card` 类基本上是两个只读字段 `suit` 和 `rank` 的容器。把字段指定为只读的原因是“空白”的牌是没有意义的，牌在创建好后也不能修改。为此，要把默认的构造函数指定为私有，并提供另一个构造函数，从给定的 `suit` 和 `rank` 中建立一副扑克牌。

此外，`Card` 类要重写 `System.Object` 的 `ToString()` 方法，这样才能获得人们可以理解的字符串，以表示扑克牌。为使编码简单一些，为两个字段 `suit` 和 `rank` 提供枚举。

`Card` 类如图 10-8 所示。

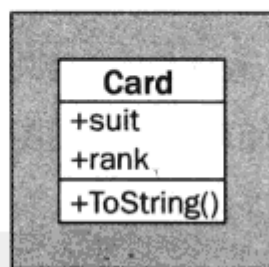


图 10-8

#### 2. Deck 类

`Deck` 类包含 52 个 `Card` 对象。我们为这些对象使用一个简单的数组类型。这个数组不能直接访问，因为对 `Card` 对象的访问要通过 `GetCard()` 方法来实现，该方法返回指定索引的 `Card` 对象。这个类也有一个 `Shuffle()` 方法，重新安排数组中的牌，所以它应如图 10-9 所示。

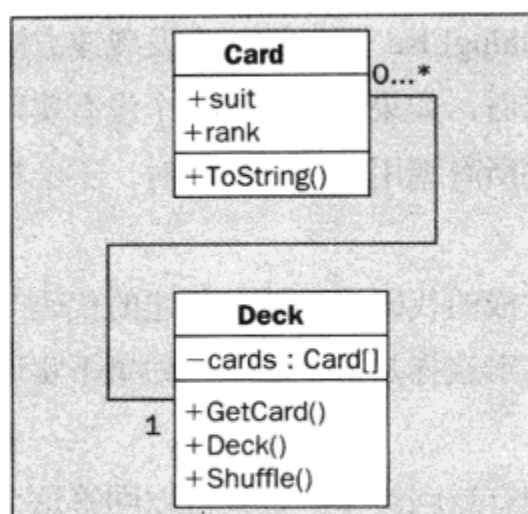


图 10-9

### 10.6.2 编写类库

对于本例，假定读者对 IDE 比较熟悉，所以不再使用标准的“试一试”方式，不再显式地列出步骤(这些步骤已经在前面多次用过)，重要的是详细讨论代码。不过，这里要包含一些指针以确保不出问题。

类和枚举都包含在一个类库项目 Ch10CardLib 中。这个项目将包含 4 个.cs 文件，Card.cs 包含 Card 类的定义，Deck.cs 包含 Deck 类的定义，Suit.cs 和 Rank.cs 文件包含枚举。

可以使用 VS 的类图工具把许多代码组合在一起。



如果使用的是 VCE，没有类图工具，不必担心。下面各节都包含了类图生成的代码，所以读者可以完成本例。这个项目的代码在不同的 IDE 中并没有区别。

首先需要完成如下操作：

- (1) 在 C:\BegVCSharp\Chapter10 目录中创建一个新类库项目 Ch10CardLib。
- (2) 从项目中删除 Class1.cs。
- (3) 在 VS 中，使用 Solution Explorer 窗口中打开项目的类图(只有选择项目，而不是选择解决方案，才能显示出类图图标)。类图开始时应为空白，因为项目不包含类。



如果在这个类图中看到 Resources 和 Settings 类，可以右击它们，选择 Remove from Diagram 选项，隐藏它们。

#### 1. 添加 Suit 和 Rank 枚举

把一个 Enum 从工具箱拖动到图中，再在显示的对话框中填充，就可以添加一个枚举。例如，对于 Suit 枚举，应在对话框中添加如图 10-10 所示的信息。



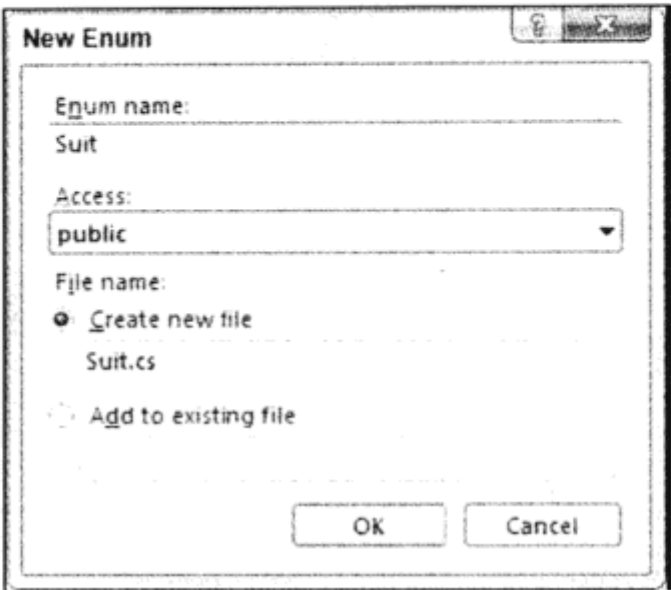


图 10-10

接着，使用 Class Details 窗口添加枚举的成员。需要添加的值如图 10-11 所示。

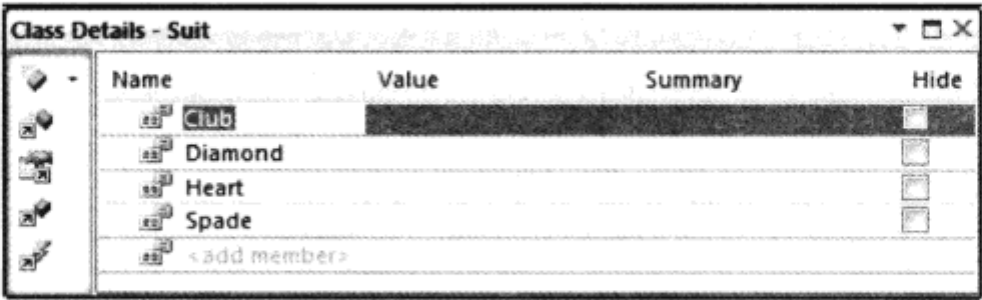


图 10-11

以相同的方式利用工具箱添加 Rank 枚举。需要的值如图 10-12 所示。

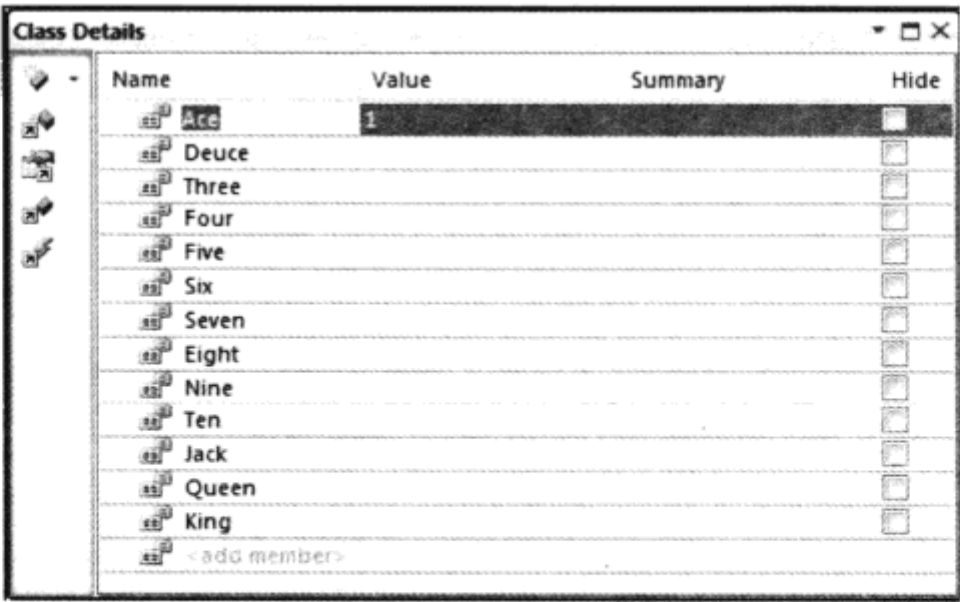


图 10-12

第一个成员 Ace 的输入值为 1，它会使枚举的底层存储匹配扑克牌的 Rank，这样 Six 就存储为 6。

完成上述操作后，类图就如图 10-13 所示。

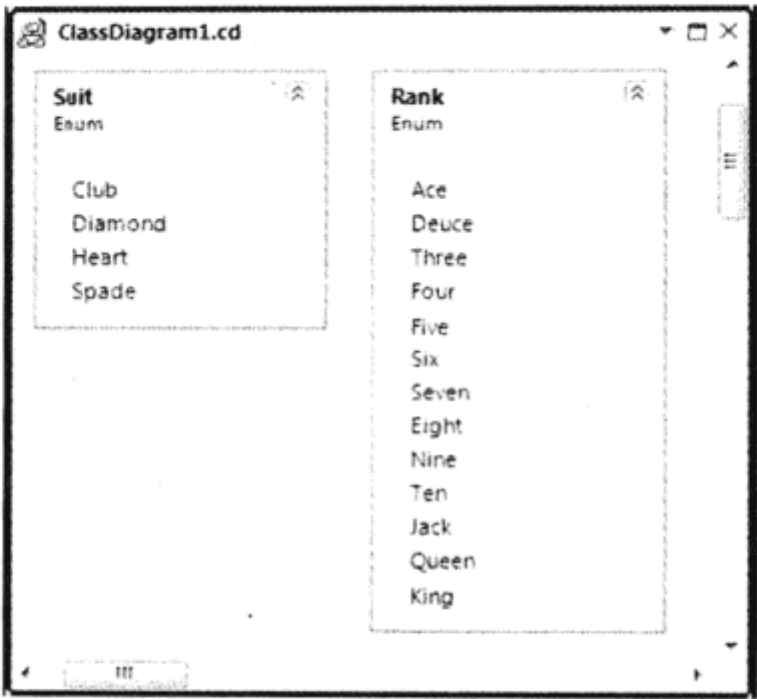


图 10-13

为这两个枚举生成的代码位于 `Suit.cs` 和 `Rank.cs` 文件中，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch10CardLib
{
    public enum Suit
    {
        Club,
        Diamond,
        Heart,
        Spade,
    }
}
```

代码段 Ch10CardLib\Suit.cs



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch10CardLib
{
    public enum Rank
    {
        Ace = 1,
        Deuce,
        Three,
        Four,
        Five,
        Six,
```

```
        Seven,  
        Eight,  
        Nine,  
        Ten,  
        Jack,  
        Queen,  
        King,  
    }  
}
```

代码段 Ch10CardLib\Rank.cs

如果使用的是 VCE，就可以添加 Suit.cs 和 Rank.cs 代码文件，再手工输入这些代码。注意，代码生成器在最后一个枚举成员后添加的逗号不会妨碍编译，不会创建一个额外的空成员，但它们可能会带来一些混乱。

2. 添加 Card 类

本节将结合使用类设计器和 VS 的代码编辑器添加 Card 类，也可以仅使用 VCE 中的代码编辑器。使用类设计器添加类与添加枚举十分类似，也是把相应的项从工具箱拖动到类图中。这里要把 Class 拖动到类图中，并把新类命名为 Card。

为了添加字段 rank 和 suit，可以使用 Class Details 窗口添加字段，再使用 Properties 窗口把字段的 Constant Kind 设置为 readonly。还需要添加两个构造函数，一个是默认构造函数(私有)，另一个构造函数带有两个参数：newSuit 和 newRank，其类型分别是 Suit 和 Rank(公共)。最后重写 ToString()，这需要在 Properties 窗口中修改 Inheritance Modifier，将它设置为 override。

图 10-14 显示了 Class Details 窗口和已输入所有信息的 Card 类。

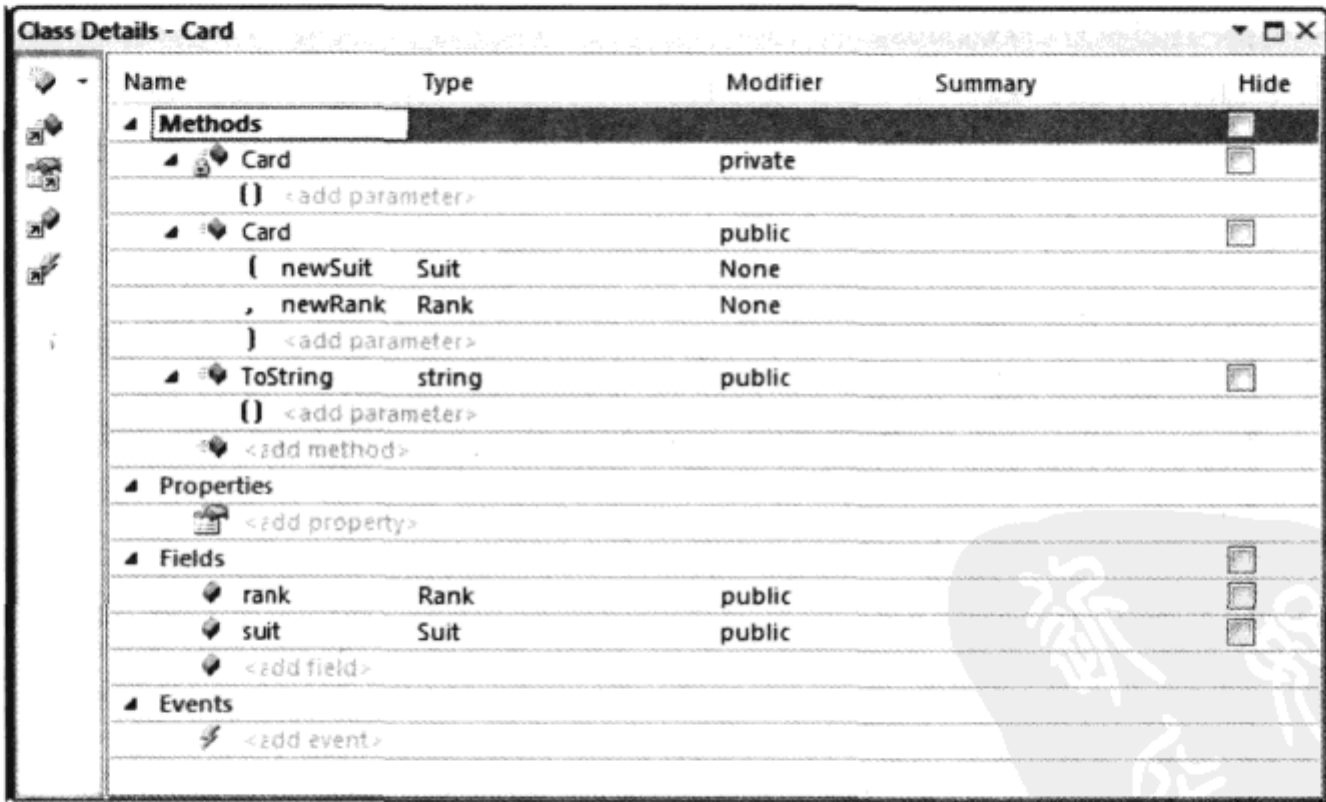


图 10-14

然后需要修改 Card.cs 中类的代码(如果使用的是 VCE，就应把这些代码添加到名称空间 Ch10CardLib 的新类 Card 中)，如下所示：



```
public class Card
{
    public readonly Suit suit;
    public readonly Rank rank;

    public Card(Suit newSuit, Rank newRank)
    {
        suit = newSuit;
        rank = newRank;
    }

    private Card()
    {
    }

    public override string ToString()
    {
        return "The " + rank + " of " + suit + "s";
    }
}
```

代码段 Ch10CardLib\Card.cs

重写的 ToString()方法将已存储的枚举值的字符串表示写入到返回的字符串中，非默认的构造函数初始化 suit 和 rank 字段的值。

3. 添加 Deck 类

Deck 类需要使用类图定义的如下成员：

- Card[]类型的私有字段 cards。
- 公共的默认构造函数。
- 公共方法 GetCard()，它带有一个 int 参数 cardNum，并返回一个 Card 类型的对象。
- 公共方法 Shuffle()，它不带参数，返回 void。

添加了这些成员后，Deck 类的 Class Details 窗口就如图 10-15 所示。

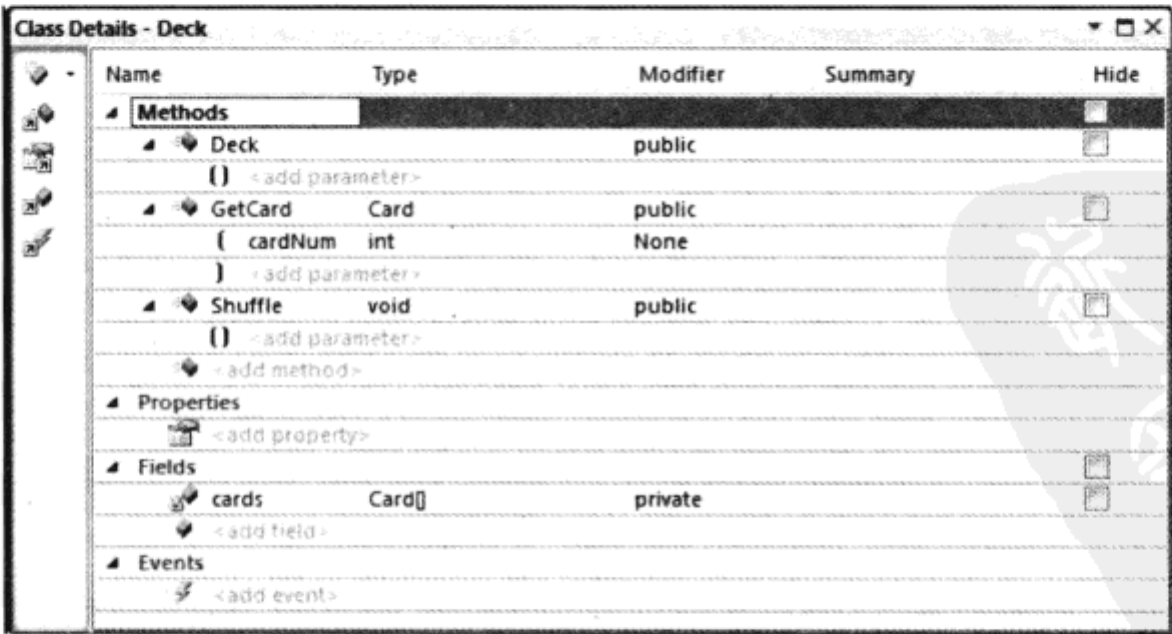


图 10-15

为了使类图更清晰，可以显示所添加的成员和类型之间的关系。在类图中依次右击下面的项，从菜单中选择 Show as Association 选项：

- Deck 中的 cards
- Card 中的 suit
- Card 中的 rank

完成后，类图如图 10-16 所示。

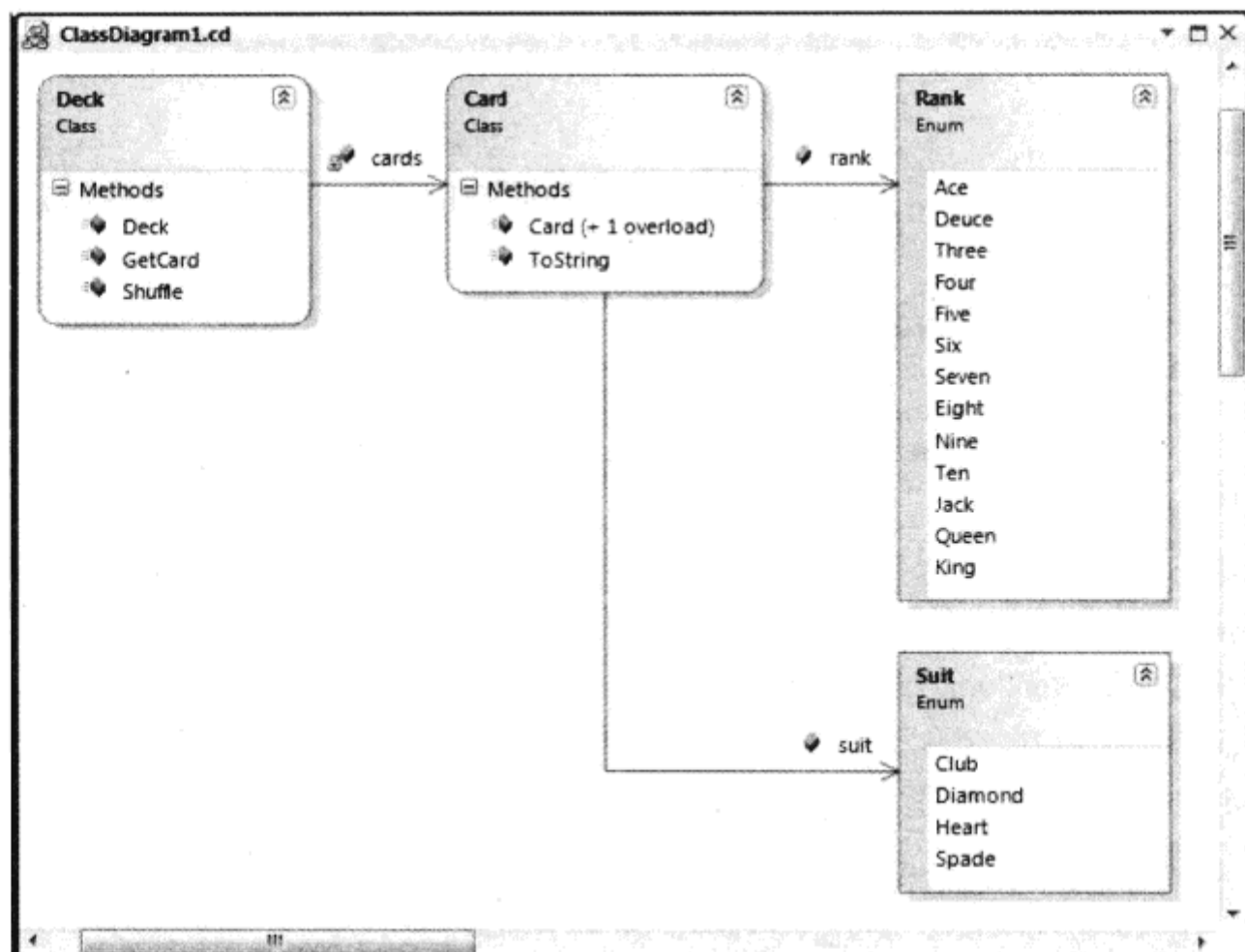


图 10-16

接着，修改 Deck.cs 中的代码(如果使用的是 VCE，就必须先使用下面的代码添加这个类)。首先实现构造函数，它在 cards 字段中创建 52 张牌，并给它们赋值。对两个枚举的所有组合进行迭代，每次迭代都创建一张牌。这将使 cards 最初包含一个有序的扑克牌列表：



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch10CardLib
{
    public Class Deck()
    {
        Private Card[] cards;
        public Deck()
        {
            cards = new Card[52];
            for (int suitVal = 0; suitVal < 4; suitVal++)
            {

```

```

        for (int rankVal = 1; rankVal < 14; rankVal++)
        {
            cards[suitVal * 13 + rankVal - 1] = new Card((Suit)suitVal,
                                                         (Rank)rankVal);
        }
    }
}

```

代码段 Ch10CardLib\Deck.cs

然后实现 GetCard()方法, 为指定的索引返回 Card 对象, 或者以与前面相同的方式抛出一个异常:

```

public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
        return cards[cardNum];
    else
        throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
                                                         "Value must be between 0 and 51."));
}

```

最后实现 Shuffle()方法。这个方法创建一个临时的扑克牌数组, 并把扑克牌从现有的 cards 数组随机复制到这个数组中。这个函数的主体是一个从 0~51 的循环, 在每次循环时, 都会使用 .NET Framework 中 System.Random 类的实例生成一个 0~51 之间的随机数。进行了实例化后, 这个类的对象使用方法 Next(X)生成一个介于 0~X 之间的随机数。有了一个随机数后, 就可以使用它作为临时数组中 Card 对象的索引, 以便复制 cards 数组中的扑克牌。

为了记录已赋值的扑克牌, 我们还有一个 bool 变量的数组, 在复制每张牌时, 把该数组中的值指定为 true。在生成随机数时, 检查这个数组, 看看是否已经把一张牌复制到临时数组中由随机数指定的位置上了, 如果已经复制好了, 就将生成另一个随机数。

这不是完成该任务的最高效的方式, 因为生成的许多随机数都可能找不到空位置以复制扑克牌。但是, 它仍能完成任务, 而且很简单, 因为 C#代码的执行速度很快, 我们几乎觉察不到延迟。代码如下:

```

public void Shuffle()
{
    Card[] newDeck = new Card[52];
    bool[] assigned = new bool[52];
    Random sourceGen = new Random();
    for (int i = 0; i < 52; i++)
    {
        int destCard = 0;
        bool foundCard = false;
        while (foundCard == false)
        {
            destCard = sourceGen.Next(52);
            if (assigned[destCard] == false)
                foundCard = true;
        }
        assigned[destCard] = true;
        newDeck[destCard] = cards[i];
    }
}

```



```

    }
    newDeck.CopyTo(cards, 0);
}
}
}

```

这个方法的最后一行使用 `System.Array` 类的 `CopyTo()` 方法(在创建数组时使用), 把 `newDeck` 中的每张扑克牌复制回 `cards` 中。也就是说, 我们使用同一个 `cards` 对象中的同一组 `Card` 对象, 而不是创建新的实例。如果改用 `cards=newDeck`, 就会用另一个对象替代 `cards` 引用的对象实例。如果其他地方的代码仍保留对原 `cards` 实例的引用, 就会出问题——不会洗牌。

至此, 就完成了类库代码。

### 10.6.3 类库的客户应用程序

为了简单起见, 可以在包含类库的解决方案中添加一个客户控制台应用程序。为此, 只需在 `Solution Explorer` 窗口中右击解决方案, 选择 `Add | New Project`, 新项目命名为 `Ch10CardClient`。

为了在这个新的控制台应用程序项目中使用前面创建的类库, 只需添加一个对类库项目 `Ch10CardLib` 的引用。为此, 可以使用 `Add Reference` 对话框的 `Projects` 选项卡, 如图 10-17 所示。

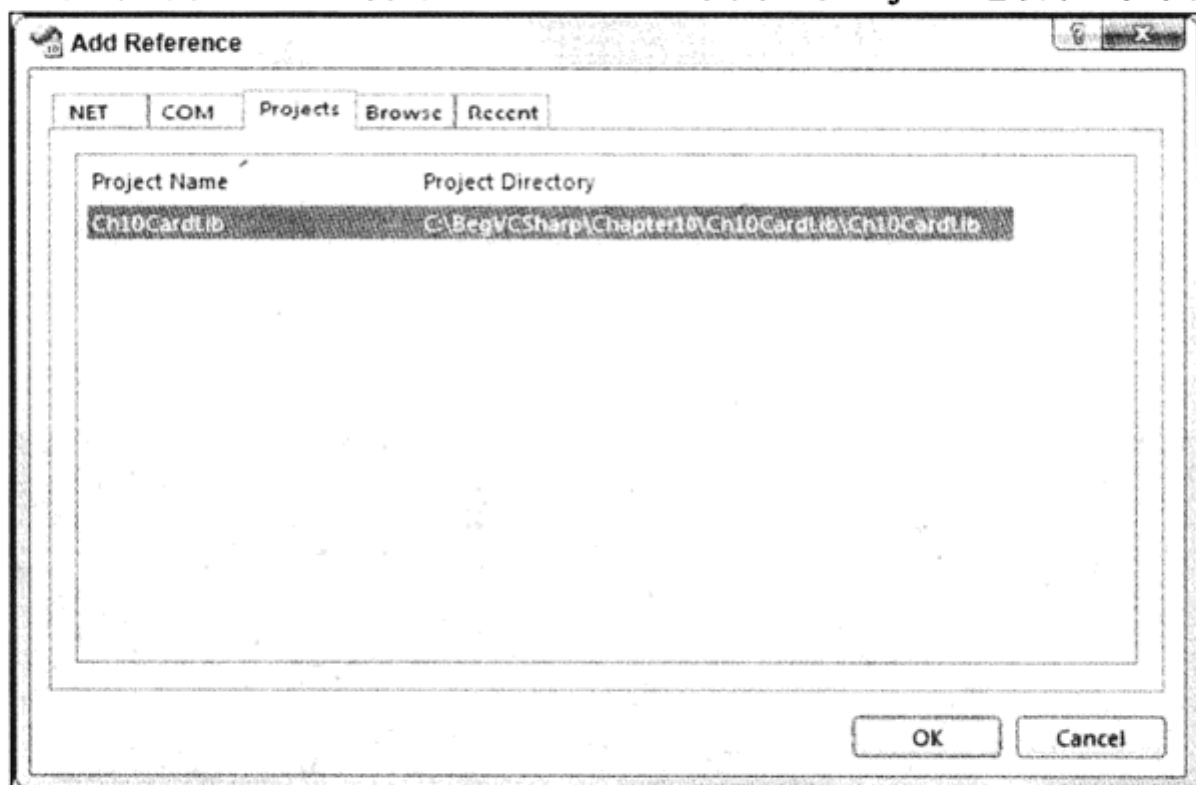



图 10-17

选择项目, 单击 `OK` 按钮, 就添加了引用。

因为这个新项目是第二个创建的, 所以还需要指定该项目为解决方案的启动项目, 即在单击 `Run` 后, 将执行这个项目。为此, 在 `Solution Explorer` 窗口中右击该项目名, 选择 `Set as StartUp Project` 菜单项。

然后需要添加使用新类的代码, 这些代码不需要做什么特别的任务, 所以添加下面的代码就可以:


 using System;  
 using System.Collections.Generic;  
 using System.Linq;  
 using System.Text;

可从  
[wrox.com](http://wrox.com)  
 下载源代码

```

using Ch10CardLib;

namespace Ch10CardClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck myDeck = new Deck();
            myDeck.Shuffle();
            for (int i = 0; i < 52; i++)
            {
                Card tempCard = myDeck.GetCard(i);
                Console.Write(tempCard.ToString());
                if (i != 51)
                    Console.Write(", ");
                else
                    Console.WriteLine();
            }
            Console.ReadKey();
        }
    }
}

```

代码段 Ch10CardClient\Program.cs

其结果如图 10-18 所示。



图 10-18

52 张扑克牌是随机放置的。后面的章节将继续开发和使用这个类库。

## 10.7 Call Hierarchy 窗口

现在分析 VS 2010 中的一项新功能：Call Hierarchy 窗口，它可以审查代码，确定方法在哪里调用，以及它们与其他方法的关系。说明这个功能的最好方式是举一个例子。

打开上一节的示例应用程序，再打开 Deck.cs 代码文件，找到 Shuffle() 方法，右击它，选择 View Call Hierarchy 菜单项，将显示如图 10-19 所示的窗口(其中展开了一些区域)。

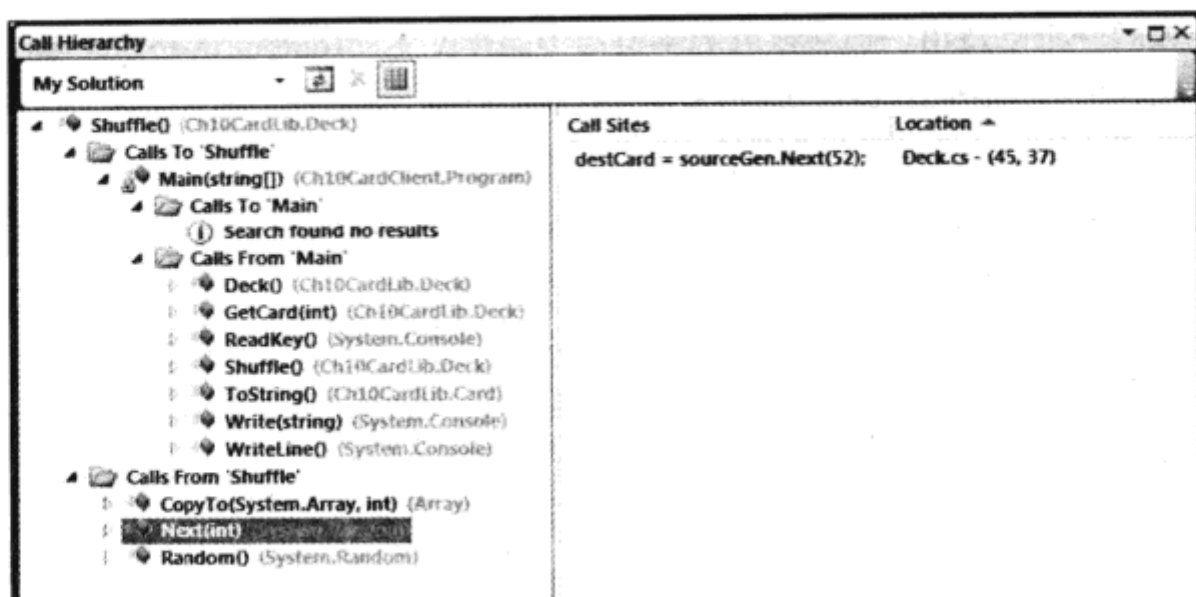


图 10-19

从 `Shuffle()` 方法开始，可以在窗口的树形视图中找出调用该方法的所有代码，以及这个方法进行的所有调用。例如，在 `Shuffle()` 中调用了图中突出显示的 `Next(int)` 方法，所以它显示在 `Calls From 'Shuffle'` 部分。单击一个调用时，会在右边看到进行这个调用的代码行及其位置。双击该位置，会立即跳到进行这个调用的代码行上。

还可以沿着层次结构向下研究其中的方法，在图 10-19 中就是 `Main()` 方法，图中显示了从 `Main()` 方法中调用的方法和调用 `Main()` 的方法。

调试和重构代码时，这个窗口是非常有用的，因为它允许查看不同部分的代码是如何相关的。

## 10.8 小结

本章结束了定义基类的讨论。仍有许多内容没有包含进来，但前面涉及到的技术已经足够创建相当复杂的应用程序了。

本章介绍了如何定义字段、方法和属性，接着讨论了各种访问级别和修饰关键字。我们还介绍了把类组合在一起的快捷工具。

介绍过这些基本主题后，我们详细讨论了继承行为，主要内容是如何用 `new` 关键字隐藏不想要的继承成员，扩展基类成员，而不是替代它们的实现代码(使用 `base` 关键字)。我们还论述了嵌套的类定义。之后，详细研究了接口的定义和实现，包括显式和隐式实现的概念。学习了如何使用部分类和部分方法定义把定义放在多个代码文件中。

最后，我们开发和使用了一个表示扑克牌的简单类库，使用方便的类图工具使工作更便于完成。后面的章节还将进一步使用这个库。

第 11 章将介绍集合，这是类的一种类型，在开发过程中经常使用。

## 10.9 练习

(1) 编写代码，定义一个基类 `MyClass`，其中包含虚拟方法 `GetString()`。这个方法应返回存储在受保护字段 `myString` 中的字符串，该字段可以通过只写公共属性 `ContainedString` 来访问。

(2) 从类 `MyClass` 中派生一个类 `MyDerivedClass`。重写 `GetString()` 方法，使用该方法的基类实现

代码从基类中返回一个字符串，但在返回的字符串中添加文本“(output from derived class)”。

(3) 部分方法定义必须使用 `void` 返回类型。说明其原因。

(4) 编写一个类 `MyCopyableClass`，该类可以使用方法 `GetCopy()` 返回它本身的一个副本。这个方法应使用派生于 `System.Object` 的 `MemberwiseClone()` 方法。给该类添加一个简单的属性，并且编写客户代码，客户代码使用该属性检查任务是否成功执行。

(5) 为 `Ch10CardLib` 库编写一个控制台客户程序，从搅乱的 `Deck` 对象中一次取出 5 张牌。如果这 5 张牌都是相同的花色，客户程序就应在屏幕上显示这 5 张牌，以及文本“Flush!”，否则就显示 50 张牌以及文本“No flush”，并退出。

附录 A 给出了练习答案。

10.10 本章要点

主 题	重 要 概 念
成员定义	可以在类中定义字段、方法和属性成员。字段用可访问性、名称和类型定义，方法用可访问性、返回类型、名称和参数定义，属性用可访问性、名称、 <code>get</code> 和/或 <code>set</code> 存取器定义。各个属性存取器可以有自己的可访问性，但它必须低于整个属性的可访问性
成员隐藏和重写	属性和方法可以在基类中定义为抽象或虚拟，以定义继承。派生类必须实现抽象的成员，使用 <code>override</code> 关键字可以重写虚拟的成员。派生类还可以用 <code>new</code> 关键字提供新的实现代码，用 <code>sealed</code> 关键字禁止进一步重写虚拟成员。基类的实现代码可以用 <code>base</code> 关键字调用
接口的实现	实现了接口的类必须实现该接口定义为公共的所有成员。可以隐式或显式实现接口，其中显式实现代码只能通过接口引用来使用
部分定义	使用 <code>partial</code> 关键字可以把类定义放在多个代码文件中。还可以使用 <code>partial</code> 关键字创建部分方法。部分方法有一些限制，包括没有返回值或 <code>out</code> 参数，如果没有提供实现代码，部分方法就不能编译

# 第 11 章

## 集合、比较和转换

### 本章内容:

---

- 如何定义和使用集合
- 可以使用的不同类型的集合
- 如何比较类型，如何使用 `is` 运算符
- 如何比较值，如何重载运算符
- 如何定义和使用转换
- 如何使用 `as` 运算符

前面讨论了 C# 中所有的基本 OOP 技术，读者还应熟悉一些比较高级的技术。本章的主要内容如下：

- **集合：**可以使用集合来维护对象组。与前面章节使用的数组不同，集合可以包含更高级的功能，例如，控制对它们包含的对象的访问、搜索和排序等。本章将介绍如何使用和创建集合类，学习掌握它们的一些强大技术。
- **比较：**在处理对象时，常常要比较它们。这对于集合尤其重要，因为这是排序的实现方式。本章将介绍如何以各种方式比较对象，包括运算符重载，使用 `Comparable` 和 `Comparer` 接口对集合排序。
- **转换：**在前面的章节中，介绍了如何把对象从一种类型转换为另一种类型。本章讨论如何定制类型转换，以满足自己的要求。

### 11.1 集合

第 5 章介绍了如何使用数组创建包含许多对象或值的变量类型。但数组有一定的限制。最大的限制是一旦创建好数组，它们的大小就是固定的，不能在现有数组的末尾添加新项，除非创建一个新的数组。这常常意味着用于处理数组的语法比较复杂。OOP 技术可以创建在内部执行大多数此类处理的类，因此简化了使用项列表或数组的代码。

C#中的数组实现为 `System.Array` 类的实例，它们只是集合类(Collection Classes)中的一种类型。集合类一般用于处理对象列表，其功能比简单数组要多，功能大多是通过实现 `System.Collections` 名称空间中的接口而获得的，因此集合的语法已经标准化了。这个名称空间还包含其他一些有趣的东西，例如，以与 `System.Array` 不同的方式实现这些接口的类。

集合的功能(包括基本功能，例如，用[index]语法访问集合中的项)可以通过接口来实现，该接口不仅没有限制我们使用基本集合类，例如`System.Array`，相反，我们还可以创建自己的定制集合类。这些集合可以专用于要枚举的对象(即要从中建立集合的对象)。这么做的一个优点是定制的集合类可以是强类型化的。也就是说，从集合中提取项时，不需要把它们转换为正确的类型。另一个优点是提供专用的方法，例如，可以提供获得项子集的快捷方法，在扑克牌示例中，可以添加一个方法，来获得特定花色中的所有 `Card` 项。

`System.Collections` 名称空间中的几个接口提供了基本的集合功能：

- `IEnumerable` 可以迭代集合中的项。
- `ICollection`(继承于 `IEnumerable`)可以获取集合中项的个数，并能把项复制到一个简单的数组类型中。
- `IList` (继承于 `IEnumerable` 和 `ICollection`)提供了集合的项列表，允许访问这些项，并提供其他一些与项列表相关的基本功能。
- `IDictionary`(继承于 `IEnumerable` 和 `ICollection`)类似于 `IList`，但提供了可通过键值(而不是索引)访问的项列表。

`System.Array` 类实现了 `IList`、`ICollection` 和 `IEnumerable`，但不支持 `IList` 的一些更高级的功能，它表示大小固定的项列表。

### 11.1.1 使用集合

`System.Collections` 名称空间中的类 `System.Collections.ArrayList` 也实现了 `IList`、`ICollection` 和 `IEnumerable` 接口，但实现方式比 `System.Array` 更复杂。数组的大小是固定的(不能增加或删除元素)，而这个类可以用于表示大小可变的项列表。为了更准确地理解这个高级集合的功能，下面介绍一个使用这个类和一个简单数组的示例。

#### 试一试：数组和高级集合

- (1) 在 `C:\BegVCSharp\Chapter11` 目录中创建一个新的控制台应用程序 `Ch11Ex01`。
- (2) 在 `Solution Explorer` 窗口中右击项目，选择 `Add | Class` 选项，给项目添加 3 个新类：`Animal`、`Cow` 和 `Chicken`。
- (3) 修改 `Animal.cs` 中的代码，如下所示：



```
namespace Ch11Ex01
{
    public abstract class Animal
    {
        protected string name;

        public string Name
        {
            get
```



```

        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    public Animal()
    {
        name = "The animal with no name";
    }

    public Animal(string newName)
    {
        name = newName;
    }

    public void Feed()
    {
        Console.WriteLine("{0} has been fed.", name);
    }
}

```

---

代码段 Ch11Ex01\Animal.cs

---

(4) 修改 Cow.cs 中的代码，如下所示：



可从  
WROX.COM  
下载源代码

```

namespace Ch11Ex01
{
    public class Cow : Animal
    {
        public void Milk()
        {
            Console.WriteLine("{0} has been milked.", name);
        }

        public Cow(string newName) : base(newName)
        {
        }
    }
}

```

---

代码段 Ch11Ex01\Cow.cs

---

(5) 修改 Chicken.cs 中的代码，如下所示：



可从  
WROX.COM  
下载源代码

```

namespace Ch11Ex01
{
    public class Chicken : Animal
    {
        public void LayEgg()
        {
        }
    }
}

```

```

        Console.WriteLine("{0} has laid an egg.", name);
    }

    public Chicken(string newName) : base(newName)
    {
    }
}

```

代码段 Ch11Ex01\Chicken.cs

(6) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Create an Array type collection of Animal " +
                              "objects and use it:");

            Animal[] animalArray = new Animal[2];
            Cow myCow1 = new Cow("Deirdre");
            animalArray[0] = myCow1;
            animalArray[1] = new Chicken("Ken");

            foreach (Animal myAnimal in animalArray)
            {
                Console.WriteLine("New {0} object added to Array collection, " +
                                  "Name = {1}", myAnimal.ToString(), myAnimal.Name);
            }

            Console.WriteLine("Array collection contains {0} objects.",
                              animalArray.Length);
            animalArray[0].Feed();
            ((Chicken)animalArray[1]).LayEgg();
            Console.WriteLine();

            Console.WriteLine("Create an ArrayList type collection of Animal " +
                              "objects and use it:");
            ArrayList animalArrayList = new ArrayList();
            Cow myCow2 = new Cow("Hayley");
            animalArrayList.Add(myCow2);
            animalArrayList.Add(new Chicken("Roy"));

            foreach (Animal myAnimal in animalArrayList)
            {
                Console.WriteLine("New {0} object added to ArrayList collection, " +

```

```

        " Name = {1}", myAnimal.ToString(), myAnimal.Name);
    }
    Console.WriteLine("ArrayList collection contains {0} objects.",
        animalArrayList.Count);
    ((Animal) animalArrayList[0]).Feed();
    ((Chicken) animalArrayList[1]).LayEgg();
    Console.WriteLine();

    Console.WriteLine("Additional manipulation of ArrayList:");
    animalArrayList.RemoveAt(0);
    ((Animal) animalArrayList[0]).Feed();
    animalArrayList.AddRange(animalArray);
    ((Chicken) animalArrayList[2]).LayEgg();
    Console.WriteLine("The animal called {0} is at index {1}.",
        myCow1.Name, animalArrayList.IndexOf(myCow1));
    myCow1.Name = "Janice";
    Console.WriteLine("The animal is now called {0}.",
        ((Animal) animalArrayList[1]).Name);
    Console.ReadKey();
}
}
}

```

代码段 Ch11Ex01\Program.cs

(7) 运行该应用程序，其结果如图 11-1 所示。

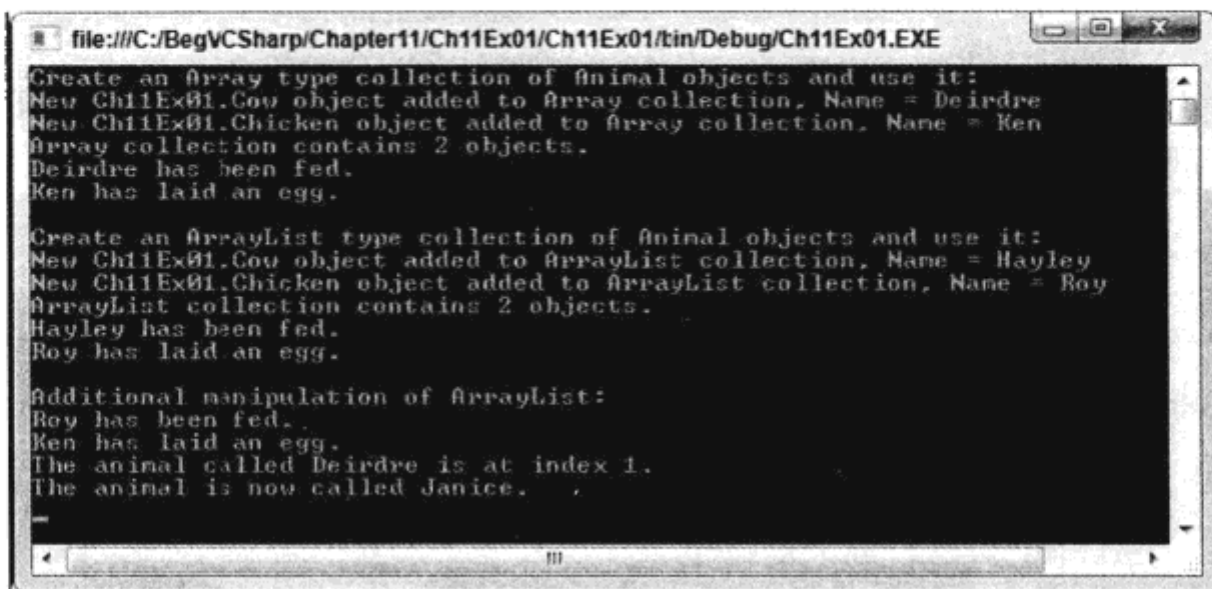


图 11-1

### 示例的说明

这个示例创建了两个对象集合，第一个集合使用 `System.Array` 类(这是一个简单的数组)，第二个集合使用 `System.Collections.ArrayList` 类。这两个集合都是 `Animal` 对象，在 `Animal.cs` 中定义。`Animal` 类是抽象类，所以不能进行实例化。但通过多态性(详见第 8 章)，可以使集合中的项成为派生于 `Animal` 类的 `Cow` 和 `Chicken` 类实例。

这些数组在 `Class1.cs` 的 `Main()` 方法中创建好后，就可以显示其特性和功能。有几个处理操作可以应用到 `Array` 和 `ArrayList` 集合上，但它们的语法略有区别。也有一些操作只能使用更高级的 `ArrayList` 类型。

下面首先通过比较这两种集合类型的代码和结果，讨论一下类似的操作。首先是集合的创建。

对于简单的数组来说，只有用固定的大小来初始化数组，才能使用它。下面使用第 5 章介绍的标准语法创建数组 `animalArray`：

```
Animal[] animalArray = new Animal[2];
```

而 `ArrayList` 集合不需要初始化其大小，所以可以使用以下代码创建列表 `animalArrayList`：

```
ArrayList animalArrayList = new ArrayList();
```

这个类还有另外两个构造函数。第一个构造函数把现有的集合作为一个参数，把现有集合的内容复制到新实例中；而另一个构造函数通过一个参数设置集合的容量(capacity)。这个容量用一个 `int` 值指定，设置集合中可以包含的初始项数。但这并不是真实的容量，因为如果集合中的项数超过了这个值，容量就会自动增加一倍。

因为数组是引用类型(例如，`Animal` 和 `Animal` 派生的对象)，所以用一个长度初始化数组并没有初始化它所包含的项。要使用一个指定的项，该项还需要初始化，即需要给这个项赋予初始化了的对象：

```
Cow myCow1 = new Cow("Deirdre");
animalArray[0] = myCow1;
animalArray[1] = new Chicken("Ken");
```

这段代码以两种方式完成该初始化任务：用现有的 `Cow` 对象来赋值，或者通过创建一个新的 `Chicken` 对象来赋值。主要的区别是前者引用了数组中的对象——我们在代码的后面就使用了这种方式。

对于 `ArrayList` 集合，它没有现成的项，也没有 `null` 引用的项。这样就不能以相同的方式给索引赋予新实例。我们使用 `ArrayList` 对象的 `Add()` 方法添加新项：

```
Cow myCow2 = new Cow("Hayley");
animalArrayList.Add(myCow2);
animalArrayList.Add(new Chicken("Roy"));
```

除了语法稍有不同外，还可以采用相同的方式把新对象或现有的对象添加到集合中。以这种方式添加完项后，就可以使用与数组相同的语法来改写它们，例如：

```
animalArrayList[0] = new Cow("Alma");
```

但不能在这个示例中这么做。

第 5 章介绍了如何使用 `foreach` 结构迭代一个数组。这是可以的，因为 `System.Array` 类实现了 `IEnumerable` 接口，这个接口的唯一方法 `GetEnumerator()` 可以迭代集合中的各项。后面将更加深入地讨论这一点。在代码中，我们写出了数组中每个 `Animal` 对象的信息：

```
foreach (Animal myAnimal in animalArray)
{
    Console.WriteLine("New {0} object added to Array collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

这里使用的 `ArrayList` 对象也支持 `IEnumerable` 接口，并可以与 `foreach` 一起使用，此时语法是相同的：

```
foreach (Animal myAnimal in animalArrayList)
{
    Console.WriteLine("New {0} object added to ArrayList collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

接着，使用数组的 `Length` 属性，在屏幕上输出数组中元素的个数：

```
Console.WriteLine("Array collection contains {0} objects.",
    animalArray.Length);
```

也可以使用 `ArrayList` 集合得到相同的结果，但要使用 `Count` 属性，该属性是 `ICollection` 接口的一部分：

```
Console.WriteLine("ArrayList collection contains {0} objects.",
    animalArrayList.Count);
```

集合——无论是简单的数组，还是比较复杂的集合——都用得不多，除非它们可以访问属于它们的项。简单数组是强类型化的，可以直接访问它们所包含的项类型。所以您可以直接调用项的方法：

```
animalArray[0].Feed();
```

数组的类型是抽象类型 `Animal`，因此不能直接调用由派生类提供的方法，而必须使用数据类型转换：

```
((Chicken) animalArray[1]).LayEgg();
```

`ArrayList` 集合是 `System.Object` 对象的集合(通过多态性赋给 `Animal` 对象)，所以必须对所有的项进行数据类型转换：

```
((Animal) animalArrayList[0]).Feed();
((Chicken) animalArrayList[1]).LayEgg();
```

代码的剩余部分利用的一些 `ArrayList` 集合功能超出了 `Array` 集合的功能范围。首先，可以使用 `Remove()` 和 `RemoveAt()` 方法删除项，这两个方法是在 `ArrayList` 类中实现的 `ICollection` 接口的一部分。它们分别根据项的引用或索引从数组中删除项。本例使用后一个方法删除列表中的第一项，即 `Name` 属性为 `Hayley` 的 `Cow` 对象：

```
animalArrayList.RemoveAt(0);
```

另外，还可以使用：

```
animalArrayList.Remove(myCow2);
```

因为这个对象已经有一个本地引用了，所以可以通过 `Add()` 添加对数组的一个现有引用，而不是创建一个新对象。无论采用哪种方式，集合中唯一剩下的项是 `Chicken` 对象，可以通过以下方式访问它：

```
((Animal) animalArrayList[0]).Feed();
```

对 `ArrayList` 对象中的项进行修改，使数组中剩下 `N` 个项，其实现方式与保留从 `0~N-1` 的索引

相同。例如，删除索引为 0 的项，会使其他项在数组中移动一个位置，所以应使用索引 0(而非 1)来访问 `Chicken` 对象。不再有索引为 1 的项了(因为集合中最初只有两个项)，所以如果试图执行下面的代码，就会抛出异常：

```
((Animal)animalArrayList[1]).Feed();
```

`ArrayList` 集合可以用 `AddRange()` 方法一次添加好几个项。这个方法接受带有 `ICollection` 接口的任何对象，包括前面的代码所创建的 `animalArray` 数组：

```
animalArrayList.AddRange(animalArray);
```

为了确定这是否有效，可以试着访问集合中的第三项，它将是 `animalArray` 中的第二项：

```
((Chicken)animalArrayList[2]).LayEgg();
```

`AddRange()` 方法不是 `ArrayList` 提供的任何接口的一部分。这个方法专用于 `ArrayList` 类，论证了可以在集合类中执行定制操作，而不仅仅是前面介绍的接口要求的操作。这个类还提供了其他有趣的方法，如 `InsertRange()`，它可以把数组对象插入到列表中的任何位置，还有用于排序和重新排序数组的方法。

最后，再回头来看看对同一个对象进行多个引用。使用 `ICollection` 接口中的 `IndexOf()` 方法可以看出，`myCow1`(最初添加到 `animalArray` 中的一个对象)现在是 `animalArrayList` 集合的一部分，它的索引如下：

```
Console.WriteLine("The animal called {0} is at index {1}.",  
    myCow1.Name, animalArrayList.IndexOf(myCow1));
```

例如，接下来的两行代码通过对象引用重新命名了对象，并通过集合引用显示了新名称：

```
myCow1.Name = "Janice";  
Console.WriteLine("The animal is now called {0}.",  
    ((Animal)animalArrayList[1]).Name);
```

### 11.1.2 定义集合

前面介绍了使用高级集合类能完成什么任务，下面讨论如何创建自己的、强类型化的集合。一种方式是手动执行需要的方法，但这比较费时间，在某些情况下也非常复杂。我们还可以从一个类中派生自己的集合，例如 `System.Collections.CollectionBase` 类，这个抽象类提供了集合类的许多实现方式。这是推荐使用的方式。

`CollectionBase` 类有接口 `IEnumerable`、`ICollection` 和 `ICollection`，但只提供了一些需要的实现代码，特别是 `ICollection` 的 `Clear()` 和 `RemoveAt()` 方法，以及 `ICollection` 的 `Count` 属性。如果要使用提供的功能，就需要自己执行其他代码。

为便于完成任务，`CollectionBase` 提供了两个受保护的属性，它们可以访问存储的对象本身。我们可以使用 `List` 和 `InnerList`，`List` 可以通过 `ICollection` 接口访问项，`InnerList` 则是用于存储项的 `ArrayList` 对象。

例如，存储 `Animal` 对象的集合类可以定义如下(稍后介绍一个比较完整的实现代码)：

```
public class Animals : CollectionBase  
{
```



```

public void Add(Animal newAnimal)
{
    List.Add(newAnimal);
}

public void Remove(Animal oldAnimal)
{
    List.Remove(oldAnimal);
}

public Animals()
{
}
}

```

其中, `Add()`和 `Remove()`方法实现为强类型化的方法, 使用 `IList` 接口中用于访问项的标准 `Add()` 方法。该方法现在只用于处理 `Animal` 类或派生于 `Animal` 的类, 而前面介绍的 `ArrayList` 实现代码可处理任何对象。

`CollectionBase` 类可以对派生的集合使用 `foreach` 语法。例如, 可以使用下面的代码:

```

Console.WriteLine("Using custom collection class Animals:");
Animals animalCollection = new Animals();
animalCollection.Add(new Cow("Sarah"));
foreach (Animal myAnimal in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}

```

但不能使用下面的代码:

```

animalCollection[0].Feed();

```

要以这种方式通过索引来访问项, 就需要使用索引符。

### 11.1.3 索引符

索引符(indexer)是一种特殊类型的属性, 可以把它添加到一个类中, 以提供类似于数组的访问。实际上, 可以通过索引符提供更复杂的访问, 因为我们可以用方括号语法定义和使用复杂的参数类型。它最常见的一个用法是对项执行简单的数字索引。

在 `Animal` 对象的 `Animals` 集合中添加一个索引符, 如下所示:

```

public class Animals : CollectionBase
{
    ...

    public Animal this[int animalIndex]
    {
        get
        {
            return (Animal)List[animalIndex];
        }
        set
    }
}

```

```

    {
        List[animalIndex] = value;
    }
}

```

**this** 关键字与方括号中的参数一起使用，但这看起来类似于其他属性。这个语法是合理的，因为在访问索引符时，将使用对象名，后跟放在方括号中的索引参数(例如 `MyAnimals[0]`)。

这段代码对 `List` 属性使用一个索引符(即在 `ICollection` 接口上，可以访问 `CollectionBase` 中的 `ArrayList`，`ArrayList` 存储了项)：

```
return (Animal)List[animalIndex];
```

这里需要进行显式数据类型转换，因为 `ICollection.List` 属性返回一个 `System.Object` 对象。注意，我们为这个索引符定义了一个类型。使用该索引符访问某项时，就可以得到这个类型。这种强类型化功能意味着，可以编写下述代码：

```
animalCollection[0].Feed();
```

而不是：

```
((Animal)animalCollection[0]).Feed();
```

这是强类型化的定制集合的另一个方便特性。下面扩展上一个示例，实践一下该特性。

#### 试一试：实现 Animals 集合

- (1) 在 `C:\BegVCSharp\Chapter11` 目录中创建一个新控制台应用程序 `Ch11Ex02`。
- (2) 在 `Solution Explorer` 窗口中右击项目名，选择 `Add | Existing Item` 选项。
- (3) 从 `C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01` 目录中选择 `Animal.cs`、`Cow.cs` 和 `Chicken.cs` 文件，单击 `Add` 按钮。
- (4) 修改这 3 个文件中的名称空间声明，如下所示：



可从  
wrox.com  
下载源代码

```
namespace Ch11Ex02
```

代码段 `Ch11Ex02\Animal.cs`、`Ch11Ex02\Cow.cs` 和 `Ch11Ex02\Chicken.cs`

- (5) 添加一个新类 `Animals`。
- (6) 修改 `Animals.cs` 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex02
{
    public class Animals : CollectionBase
    {
        public void Add(Animal newAnimal)
    }
}

```

```

    {
        List.Add(newAnimal);
    }

    public void Remove(Animal newAnimal)
    {
        List.Remove(newAnimal);
    }

    public Animals()
    {
    }

    public Animal this[int animalIndex]
    {
        get
        {
            return (Animal)List[animalIndex];
        }
        set
        {
            List[animalIndex] = value;
        }
    }
}
}

```

代码段 Ch11Ex02\Animals.cs

(7) 修改 Program.cs, 如下所示:



可从  
wrox.com  
下载源代码

```

static void Main(string[] args)
{
    Animals animalCollection = new Animals();
    animalCollection.Add(new Cow("Jack"));
    animalCollection.Add(new Chicken("Vera"));
    foreach (Animal myAnimal in animalCollection)
    {
        myAnimal.Feed();
    }
    Console.ReadKey();
}

```

代码段 Ch11Ex02\Program.cs

(8) 执行应用程序, 其结果如图 11-2 所示。

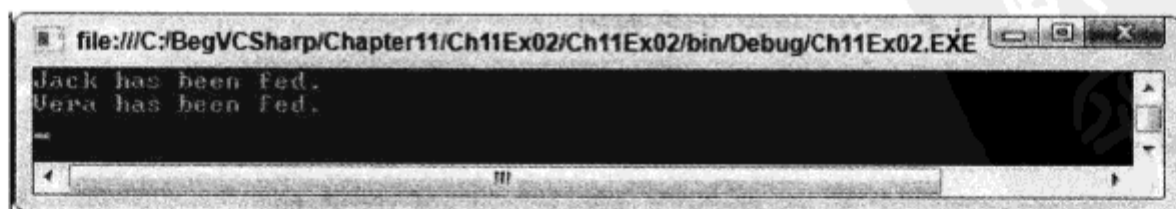


图 11-2

### 示例的说明

这个示例使用上一节详细介绍的代码，实现类 `Animals` 中强类型化的 `Animal` 对象集合。`Main()` 中的代码仅实例化了一个 `Animals` 对象 `animalCollection`，添加了两个项(它们分别是 `Cow` 和 `Chicken` 的实例)，再使用 `foreach` 循环调用这两个对象继承于基类 `Animal` 的 `Feed()` 方法。

#### 11.1.4 给 CardLib 添加 Cards 集合

第 10 章创建了一个类库项目 `Ch10CardLib`，它包含一个表示扑克牌的 `Card` 类和一个表示一幅扑克牌的 `Deck` 类，这个 `Deck` 类是 `Card` 类的集合，且实现为一个简单的数组。

本章给这个库添加一个新类，并把类库重命名为 `Ch11CardLib`。这个新类 `Cards` 是 `Card` 对象的一个定制集合，并拥有本章前面介绍的各种功能。在 `C:\BegVCSharp\Chapter11` 目录中创建一个新的类库 `Ch11CardLib`，再从 `Project | Add Existing Item` 中选择 `C:\BegVCSharp\Chapter10\Ch10CardLib\Ch10CardLib` 目录中的 `Card.cs`、`Deck.cs`、`Suit.cs` 和 `Rank.cs` 文件，把它们添加到项目中。与第 10 章介绍的这个项目的上一个版本相同，这里也不使用标准的“试一试”格式介绍这些变化。读者可以在本章的下载代码中打开这个项目的版本，直接查看代码。



在把源文件从 `Ch10CardLib` 复制到 `Ch11CardLib` 中时，必须修改名称空间声明，以引用 `Ch11CardLib`。对用于测试的 `Ch10CardLib` 控制台应用程序也要进行这个修改。

本章下载代码中的一个项目包含了对 `Ch11CardClient` 进行的各种扩展。其代码放在各个区段中，如果读者要实践一下，可以取消这些区段的注释。

如果要自己创建这个项目，就应添加一个新类 `Cards`，修改 `Cards.cs` 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
    public class Cards : CollectionBase
    {
        public void Add(Card newCard)
        {
            List.Add(newCard);
        }

        public void Remove(Card oldCard)
        {
            List.Remove(oldCard);
        }

        public Cards()
        {
        }
    }
}
```

```

public Card this[int cardIndex]
{
    get
    {
        return (Card)List[cardIndex];
    }
    set
    {
        List[cardIndex] = value;
    }
}

/// <summary>
/// Utility method for copying card instance into another Cards
/// instance - used in Deck.Shuffle(). This implementation assume that
/// source and target collections are the same size.
/// </summary>
public void CopyTo(Cards targetCards)
{
    for (int index = 0; index < this.Count; index++)
    {
        targetCards[index] = this[index];
    }
}

/// <summary>
/// Check to see if the Cards collection contains a particular card.
/// This calls the Contains method of the ArrayList for the collection,
/// which we access through the InnerList property.
/// </summary>
public bool Contains(Card card)
{
    return InnerList.Contains(card);
}
}

```

代码段 Ch11CardLib\Cards.cs

然后，需要修改 Deck.cs，以利用这个新集合，而不是数组：



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
    public class Deck
    {
        private Cards cards = new Cards();

        public Deck()
        {
            // line of code removed here.
        }
    }
}

```

```

        for (int suitVal = 0; suitVal < 4; suitVal++)
        {
            for (int rankVal = 1; rankVal < 14; rankVal++)
            {
                cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
            }
        }
    }

    public Card GetCard(int cardNum)
    {
        if (cardNum >= 0 && cardNum <= 51)
            return cards[cardNum];
        else
            throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
                "Value must be between 0 and 51."));
    }

    public void Shuffle()
    {
        Cards newDeck = new Cards();
        bool[] assigned = new bool[52];
        Random sourceGen = new Random();
        for (int i = 0; i < 52; i++)
        {
            int sourceCard = 0;
            bool foundCard = false;
            while (foundCard == false)
            {
                sourceCard = sourceGen.Next(52);
                if (assigned[sourceCard] == false)
                {
                    foundCard = true;
                }
            }
            assigned[sourceCard] = true;
            newDeck.Add(cards[sourceCard]);
        }
        newDeck.CopyTo(cards);
    }
}

```

代码段 Ch11CardLib\Deck.cs

在此不需要进行很多修改。其中的大多数修改都涉及到改变洗牌逻辑，才能把 Cards 中随机的一张牌添加到新 Cards 集合 newDeck 的开头，而不是把 cards 集合中顺序位置的一张牌添加 newDeck 集合的随机位置上。

Ch10CardLib 解决方案的客户控制台应用程序 Ch10CardClient 可以使用这个新库得到与以前相同的结果，因为 Deck 的方法签名没有改变。这个类库的客户程序现在可以使用 Cards 集合类，而不是依赖 Card 对象数组，例如，在扑克牌游戏应用程序中定义一手牌。

### 11.1.5 关键字值集合和 IDictionary

除了 IList 接口外，集合还可以实现类似的 IDictionary 接口，允许项通过关键字值(如字符串名)



进行索引，而不是通过一个索引。这也可以使用索引符来完成，但这次的索引符参数是与存储的项相关联的关键字，而不是 `int` 索引，这样集合就更便于用户使用了。

与索引的集合一样，可以使用一个基类简化 `IDictionary` 接口的实现，这个基类就是 `DictionaryBase`，它也实现 `IEnumerable` 和 `ICollection`，提供了对任何集合都相同的基本集合处理功能。

`DictionaryBase` 与 `CollectionBase` 一样，实现通过其支持的接口获得的一些成员(但不是全部成员)。`DictionaryBase` 也实现 `Clear` 和 `Count` 成员，但不实现 `RemoveAt()`。这是因为 `RemoveAt()` 是 `ICollection` 接口中的一个方法，不是 `IDictionary` 接口中的一个方法。但是，`IDictionary` 有一个 `Remove()` 方法，这是一个应在基于 `DictionaryBase` 的定制集合类上实现的方法。

下面的代码是 `Animals` 类的另一个版本，这次该类派生于 `DictionaryBase`。下面代码包括 `Add()`、`Remove()` 和一个通过关键字访问的索引符的实现代码：

```
public class Animals : DictionaryBase
{
    public void Add(string newID, Animal newAnimal)
    {
        Dictionary.Add(newID, newAnimal);
    }

    public void Remove(string animalID)
    {
        Dictionary.Remove(animalID);
    }

    public Animals()
    {
    }

    public Animal this[string animalID]
    {
        get
        {
            return (Animal)Dictionary[animalID];
        }
        set
        {
            Dictionary[animalID] = value;
        }
    }
}
```

这些成员的区别如下：

- **Add()**——带有两个参数：一个关键字和一个值，存储在一起。字典集合有一个继承于 `DictionaryBase` 的成员 `Dictionary`，这个成员是一个 `IDictionary` 接口，有自己的 `Add()` 方法，该方法带有两个 `object` 参数。我们的实现代码带有一个 `string` 值(作为关键字)和一个 `Animal` 对象(作为与该关键字存储在一起的数据)。
- **Remove()**——带有一个关键字参数，而不是对象引用。带有指定关键字值的项被删除。

- **Indexer**——使用一个字符串关键字值，而不是一个索引，用于通过Dictionary 的继承成员来访问存储的项，这里仍需要进行数据类型转换。

基于 DictionaryBase 的集合和基于 CollectionBase 的集合之间的另一个区别是 foreach 的工作方式略有区别。上一节的集合可以直接从集合中提取 Animal 对象。使用 foreach 和 DictionaryBase 派生类可以提供 DictionaryEntry 结构，这是在 System.Collections 名称空间中定义的另一个类型。要得到 Animal 对象本身，就必须使用这个结构的 Value 成员，也可以使用结构的 Key 成员得到相关的关键字。要使代码等价于前面的代码：

```
foreach (Animal myAnimal in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name());
}
```

需要使用以下代码：

```
foreach (DictionaryEntry myEntry in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myEntry.Value.ToString(),
        ((Animal)myEntry.Value).Name);
}
```

重写这段代码，以便直接通过 foreach 提取 Animal 对象，这有许多方式，最简单的方式是实现一个迭代器。

### 11.1.6 迭代器

本章前面介绍过，IEnumerable 接口负责使用 foreach 循环。在 foreach 循环中使用定制类通常有很多优点，而不仅仅使用集合类，例如，本章前面几节介绍的集合类。

但是，重写使用 foreach 循环的方式，或者提供定制的实现方式，并不一定很简单。为了说明这一点，下面深入研究一下 foreach 循环。在 foreach 循环中，迭代集合 collectionObject 的过程如下：

(1) 调用 collectionObject.GetEnumerator()，返回一个 IEnumerator 引用。这个方法可以通过 IEnumerable 接口的实现代码来获得，但这是可选的。

(2) 调用所返回的 IEnumerator 接口的 MoveNext()方法。

(3) 如果 MoveNext()方法返回 true，就使用 IEnumerator 接口的 Current 属性获取对象的一个引用，用于 foreach 循环。

(4) 重复前面两步，直到 MoveNext()方法返回 false 为止，此时循环停止。

所以，为了在类中进行这些操作，必须重写几个方法，跟踪索引，维护 Current 属性，以及执行其他一些操作，这要做许多工作。

一个较为简单的替代方法是使用迭代器。使用迭代器将有效地在后台生成许多代码，正确地完成任务。而且，使用迭代器的语法掌握起来非常容易。

迭代器的定义是，它是一个代码块，按顺序提供了要在 foreach 循环中使用的所有值。一般情况下，这个代码块是一个方法，但也可以使用属性访问器和其他代码块作为迭代器。这里为了简单起见，仅介绍方法。

无论代码块是什么，其返回类型都是有限制的。与期望正好相反，这个返回类型与所枚举的对象类型不同。例如，在表示 `Animal` 对象集合的类中，迭代器块的返回类型不可能是 `Animal`。两种可能的返回类型是前面提到的接口类型 `IEnumerable` 和 `IEnumerator`。使用这两个类型的场合是：

- 如果要迭代一个类，可使用方法 `GetEnumerator()`，其返回类型是 `IEnumerator`。
- 如果要迭代一个类成员，例如一个方法，则使用 `IEnumerable`。

在迭代器块中，使用 `yield` 关键字选择要在 `foreach` 循环中使用的值。其语法如下：

```
yield return value;
```

利用这个信息就足以建立一个非常简单的示例，如下所示：



```
public static IEnumerable SimpleList()
{
    yield return "string 1";
    yield return "string 2";
    yield return "string 3";
}

public static void Main(string[] args)
{
    foreach (string item in SimpleList())
        Console.WriteLine(item);

    Console.ReadKey();
}
```

代码段 SimpleIterators\Program.cs



为了亲手测试这些代码，应给 `System.Collections` 名称空间添加一个 `using` 语句，或者使用完全限定的 `System.Collections.IEnumerable` 接口。这些代码在本章下载代码的 `SimpleIterators` 项目中。

在此，静态方法 `SimpleList()` 就是迭代器块。它是一个方法，所以使用 `IEnumerable` 返回类型。`SimpleList()` 使用 `yield` 关键字为使用它的 `foreach` 块提供了 3 个值，每个值都输出到屏幕上，结果如图 11-3 所示。

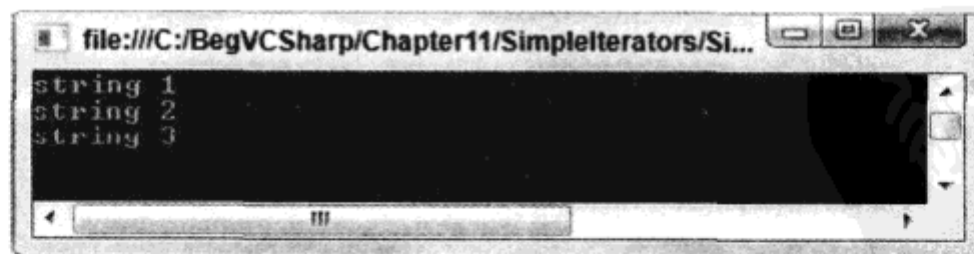


图 11-3

显然，这个迭代器并不是特别有用，但它允许查看执行过程，了解实现代码有多简单。看看代码，读者可能会疑惑代码是如何知道返回 `string` 类型的项。实际上，并没有返回 `string` 类型的项，而

是返回了 `object` 类型的值。因为 `object` 是所有类型的基类，也就是说，可以从 `yield` 语句中返回任意类型。

但是，编译器非常聪明，所以我们可以把返回值解释为 `foreach` 循环需要的任何类型。这里代码需要 `string` 类型的值，所以这就是我们要使用的值。如果修改一行 `yield` 代码，让它返回一个整数，就会在 `foreach` 循环中出现一个错误类型转换异常。

对于迭代器，还有一点要注意。可以使用下面的语句中断信息返回 `foreach` 循环的过程：

```
yield break;
```

在遇到迭代器中的这个语句时，迭代器的处理会立即中断，就像 `foreach` 循环使用它一样。

下面是一个比较复杂但很有用的示例。在这个示例中，要实现一个迭代器，获取素数。

### 试一试：实现一个迭代器

- (1) 在 `C:\BegVCSharp\Chapter11` 目录中创建一个新控制台应用程序 `Ch11Ex03`。
- (2) 添加一个新类 `Primes`，修改代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex03
{
    public class Primes
    {
        private long min;
        private long max;

        public Primes() : this(2, 100)
        {
        }

        public Primes(long minimum, long maximum)
        {
            if (min < 2)
                min = 2;
            else
                min = minimum;

            max = maximum;
        }

        public IEnumerator GetEnumerator()
        {
            for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)
            {
                bool isPrime = true;
                for (long possibleFactor = 2; possibleFactor <=
                    (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
                {

```

```

        long remainderAfterDivision = possiblePrime % possibleFactor;
        if (remainderAfterDivision == 0)
        {
            isPrime = false;
            break;
        }
    }
    if (isPrime)
    {
        yield return possiblePrime;
    }
}
}
}

```

代码段 Ch11Ex03\Primes.cs

(3) 修改 Program.cs 中的代码，如下所示：



```

static void Main(string[] args)
{
    Primes primesFrom2To1000 = new Primes(2, 1000);
    foreach (long i in primesFrom2To1000)
        Console.Write("{0} ", i);

    Console.ReadKey();
}

```

代码段 Ch11Ex03\Program.cs

(4) 执行应用程序，结果如图 11-4 所示。

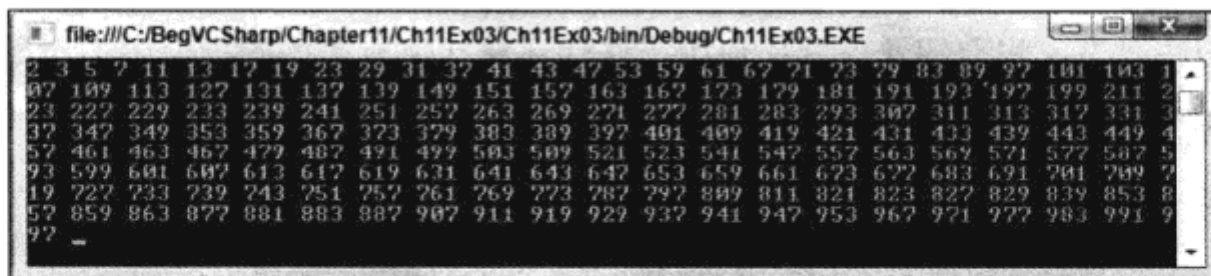


图 11-4

### 示例的说明

这个示例中的类可以枚举上下限之间的素数集合。封装素数的类利用迭代器提供了这个功能。

Primes 的代码开始时比较简单，用两个字段存储表示搜索范围的最大值和最小值，并使用构造函数设置这些值。注意，最小值是有限制的，它不能小于 2，这是有意义的，因为 2 是最小的素数。相关的代码则全部放在方法 GetEnumerator() 中。该方法的签名满足迭代器块的规则，因为它返回 IEnumerator 类型：

```

public IEnumerator GetEnumerator()
{

```

为了提取上下限之间的素数，需要依次测试每个值，所以用一个 for 循环开始：

```

    for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)
    {

```

由于我们不知道某个数是否是素数，所以先假定这个数是素数，再看看它是否是素数。为此，需要看看该数能否被 2 到该数平方根之间的所有数整除。如果能，则该数不是素数，于是测试下一个数。如果该数的确是素数，就使用 `yield` 把它传送给 `foreach` 循环。

```

        bool isPrime = true;
        for (long possibleFactor = 2; possibleFactor <=
            (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
        {
            long remainderAfterDivision = possiblePrime % possibleFactor;
            if (remainderAfterDivision == 0)
            {
                isPrime = false;
                break;
            }
        }
        if (isPrime)
        {
            yield return possiblePrime;
        }
    }
}

```

在这段代码中，有一个有趣的地方：如果把上下限设置为非常大的数，在执行应用程序时，就会发现，会一次显示一个结果，中间有暂停，而不是一次显示所有结果。这说明，无论代码在 `yield` 调用之间是否终止，迭代器代码都会一次返回一个结果。在后台，调用 `yield` 都会中断代码的执行，当请求另一个值时，也就是当使用迭代器的 `foreach` 循环开始一个新的循环时，代码会恢复执行。

### 迭代器和集合

前面我们许诺过，将介绍迭代器如何用于迭代存储在字典类型的集合中的对象，无需处理 `DictionaryItem` 对象。下面是集合类 `Animals`：

```

public class Animals : DictionaryBase
{
    public void Add(string newID, Animal newAnimal)
    {
        Dictionary.Add(newID, newAnimal);
    }

    public void Remove(string animalID)
    {
        Dictionary.Remove(animalID);
    }

    public Animals()
    {
    }

    public Animal this[string animalID]
    {

```



```

    get
    {
        return (Animal)Dictionary[animalID];
    }
    set
    {
        Dictionary[animalID] = value;
    }
}

```

可以在这段代码中添加如下简单的迭代器，以便执行预期的操作：



```

public new IEnumerator GetEnumerator()
{
    foreach (object animal in Dictionary.Values)
        yield return (Animal)animal;
}

```

代码段 DictionaryAnimals\Animals.cs

现在可以使用下面的代码迭代集合中的 `Animal` 对象了：



```

foreach (Animal myAnimal in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}

```

代码段 DictionaryAnimals\Program.cs



在本章的下载代码中，这些代码位于 `DictionaryAnimals` 项目中。

### 11.1.7 深复制

第 9 章通过下面的 `GetCopy()` 方法，介绍了如何使用受保护的方法 `System.Object.MemberwiseClone()` 进行浅复制(shallow copy)。

```

public class Cloner
{
    public int Val;

    public Cloner(int newVal)
    {
        Val = newVal;
    }

    public object GetCopy()
    {
        return MemberwiseClone();
    }
}

```

```

    }
}

```

假定有引用类型的字段，而不是值类型的字段(例如，对象)：

```

public class Content
{
    public int Val;
}

public class Cloner
{
    public Content MyContent = new Content();

    public Cloner(int newVal)
    {
        MyContent.Val = newVal;
    }

    public object GetCopy()
    {
        return MemberwiseClone();
    }
}

```

此时，通过GetCopy()得到的浅复制包括一个字段，它引用的对象与源对象相同。下面的代码使用这个 Cloner 类来说明浅复制引用类型的结果：

```

Cloner mySource = new Cloner(5);
Cloner myTarget = (Cloner)mySource.GetCopy();
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
mySource.MyContent.Val = 2;
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);

```

第 4 行把一个值赋给 mySource.MyContent.Val，它是源对象中公共字段 MyContent 的公共字段 Val。这也改变了 myTarget.MyContent.Val 的值。这是因为 mySource.MyContent 引用了与 myTarget.MyContent 相同的对象实例。上述代码的输出结果如下：

```

myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 2

```

为了解决这个问题，需要执行深复制。修改上面的 GetCopy()方法就可以进行深复制，但最好使用.NET Framework 的标准方式：实现ICloneable 接口，该接口有一个方法 Clone()；这个方法不带参数，返回一个 object 类型的结果，其签名和上面使用的 GetCopy()方法相同。

修改上面的类，以使用下面的深复制代码：

```

public class Content
{
    public int Val;
}

public class Cloner : ICloneable
{

```

```

public Content MyContent = new Content();

public Cloner(int newVal)
{
    MyContent.Val = newVal;
}

public object Clone()
{
    Cloner clonedCloner = new Cloner(MyContent.Val);
    return clonedCloner;
}
}

```

其中使用包含在源 Cloner 对象中的 Content 对象(MyContent)的 Val 字段, 创建一个新 Cloner 对象。这个字段是一个值类型, 所以不需要深复制。

使用与上面类似的代码测试浅复制, 但用 Clone() 替代 GetCopy(), 得到如下结果:

```

myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 5

```

这次包含的对象是独立的。注意有时在比较复杂的对象系统中, 调用 Clone() 是一个递归过程。例如, 如果 Cloner 类的 MyContent 字段也需要深复制, 就要使用下面的代码:

```

public class Cloner : ICloneable
{
    public Content MyContent = new Content();

    ...

    public object Clone()
    {
        Cloner clonedCloner = new Cloner();
        clonedCloner.MyContent = MyContent.Clone();
        return clonedCloner;
    }
}

```

这里调用了默认的构造函数, 以便简化创建一个新 Cloner 对象的语法。为了使这段代码能正常工作, 还需要在 Content 类上实现 ICloneable 接口。

### 11.1.8 给 CardLib 添加深复制

下面把上述内容付诸于实践: 使用 ICloneable 接口, 复制 Card、Cards 和 Deck 对象, 这在某些扑克牌游戏中是有用的, 因为在这些游戏中不需要让两副扑克牌引用一组相同的 Card 对象, 但肯定会使一副扑克牌中的牌序与另一副牌的牌序相同。

在 Ch11CardLib 中, 对 Card 类执行复制操作是很简单的, 因为只需进行浅复制(Card 只包含值类型的数据, 其形式为字段)。我们只需对类定义进行如下修改:



可从  
wrox.com  
下载源代码

```

public class Card : ICloneable
{
    public object Clone()
    {

```

```

        return MemberwiseClone();
    }

```

代码段 Ch11CardLib\Card.cs

ICloneable 接口的这段实现代码只是一个浅复制，无法确定在 Clone() 方法中执行了什么操作，而这正是我们的目的。

接着，需要对 Cards 集合类实现 ICloneable 接口。这个过程稍复杂些，因为涉及到复制源集合中的每个 Card 对象，所以需要进行深复制：



```

public class Cards : CollectionBase, ICloneable
{
    public object Clone()
    {
        Cards newCards = new Cards();
        foreach (Card sourceCard in List)
        {
            newCards.Add(sourceCard.Clone() as Card);
        }
        return newCards;
    }
}

```

代码段 Ch11CardLib\Cards.cs

最后，需要在 Deck 类上实现 ICloneable 接口。这里存在一个小问题：因为 Deck 类无法修改它包含的扑克牌，所以没有洗牌。例如，无法修改有给定牌序的 Deck 实例。为了解决这个问题，为 Deck 类定义一个新的私有构造函数，在实例化 Deck 对象时，可以给该函数传送指定的 Cards 集合。所以，在这个类中执行复制的代码如下所示：



```

public class Deck : ICloneable
{
    public object Clone()
    {
        Deck newDeck = new Deck(cards.Clone() as Cards);
        return newDeck;
    }

    private Deck(Cards newCards)
    {
        cards = newCards;
    }
}

```

代码段 Ch11CardLib\Deck.cs

再次用一些简单的客户代码进行测试(与以前一样，这应放在客户项目的 Main() 方法中，以便进行测试)：



```

Deck deck1 = new Deck();
Deck deck2 = (Deck)deck1.Clone();
Console.WriteLine("The first card in the original deck is: {0}",

```

```

        deck1.GetCard(0));
    Console.WriteLine("The first card in the cloned deck is: {0}",
        deck2.GetCard(0));

    deck1.Shuffle();
    Console.WriteLine("Original deck shuffled.");
    Console.WriteLine("The first card in the original deck is: {0}",
        deck1.GetCard(0));
    Console.WriteLine("The first card in the cloned deck is: {0}",
        deck2.GetCard(0));

    Console.ReadKey();

```

代码段 Ch11CardClient\Program.cs

其输出结果如图 11-5 所示。

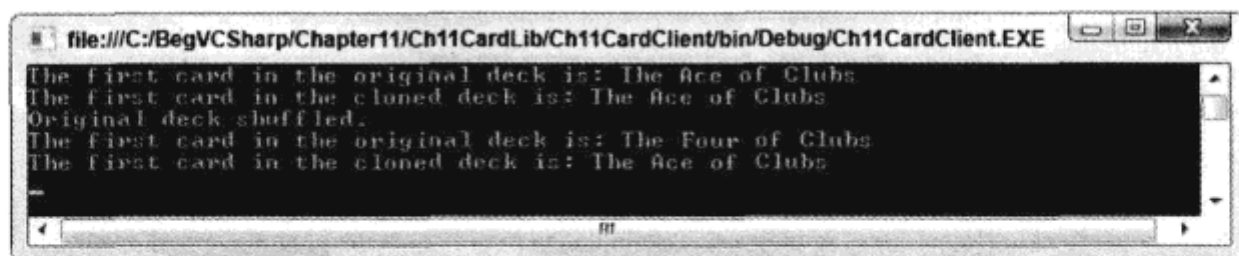


图 11-5

## 11.2 比较

本节介绍对象之间的两类比较：

- 类型比较
- 值比较

类型比较确定对象是什么，或者对象继承了什么，在 C# 编程中，这是非常重要的。把对象传送给方法时，下一步要进行什么操作常常取决于对象的类型。本章和前面的章节都讨论过传送对象的内容，这里将介绍一些更有用的技巧。

值比较我们也见过许多，至少见过简单类型的值比较。在比较对象的值时，情况会变得较为复杂：必须从一开始就定义比较的含义，确定像 > 这样的运算符在类中会执行什么操作。这在集合中尤其重要，有时我们希望根据某个条件排列对象的顺序，例如按照字母顺序或者根据某个比较复杂的算法来排序。

### 11.2.1 类型比较

在比较对象时，常常需要了解它们的类型，才能确定是否可以进行比较。第 9 章介绍了 `GetType()` 方法，所有的类都从 `System.Object` 中继承了这个方法，这个方法和 `typeof()` 运算符一起使用，就可以确定对象的类型(并据此执行操作)：

```

if (myObj.GetType() == typeof(MyComplexClass))
{
    // myObj is an instance of the class MyComplexClass.
}

```

前面还提到 ToString() 的默认实现方式, ToString() 也是从 System.Object 继承来的, 该方法可以提供对象类型的字符串表示。也可以比较这些字符串, 但这是比较杂乱的方式。

本节将介绍比较值的一种简便方式: is 运算符。它可以提供可读性较高的代码, 还可以检查基类。在介绍 is 运算符之前, 需要了解处理值类型(与引用类型相反)时后台的一些常见操作: 封箱(boxing)和拆箱(unboxing)。

## 1. 封箱和拆箱

第 8 章讨论了引用类型和值类型之间的区别, 第 9 章通过比较结构(值类型)和类(引用类型)进行了说明。封箱(boxing)是把值类型转换为 System.Object 类型, 或者转换为由值类型实现的接口类型。拆箱(unboxing)是相反的过程。

例如, 下面的结构类型:

```
struct MyStruct
{
    public int Val;
}
```

可以把这种类型的结构放在 object 类型的变量中, 以封箱它:

```
MyStruct valType1 = new MyStruct();
valType1.Val = 5;
object refType = valType1;
```

其中创建了一个类型为 MyStruct 的新变量(valType1), 并把一个值赋予这个结构的 Val 成员, 然后把它封箱在 object 类型的变量(refType)中。

以这种方式封箱变量而创建的对象, 包含值类型变量的一个副本的引用, 而不包含源值类型变量的引用。要进行验证, 可以修改源结构的内容, 把对象中包含的结构拆箱到新变量中, 检查其内容:

```
valType1.Val = 6;
MyStruct valType2 = (MyStruct)refType;
Console.WriteLine("valType2.Val = {0}", valType2.Val);
```

执行这段代码将得到如下输出结果:

```
valType2.Val = 5
```

但在把一个引用类型赋予对象时, 将执行不同的操作。把 MyStruct 改为一个类(不考虑这个类名不合适的情况), 即可看到这种情形:

```
class MyStruct
{
    public int Val;
}
```

如果不修改上面的客户代码(再次忽略名称错误的变量), 就会得到如下输出结果:

```
valType2.Val = 6
```



也可以把值类型封箱到一个接口类型中,只要它们实现这个接口即可。例如,假定 `MyStruct` 类型实现 `IMyInterface` 接口,如下所示:

```
interface IMyInterface
{
}

struct MyStruct : IMyInterface
{
    public int Val;
}
```

接着把结构封箱到一个 `IMyInterface` 类型中,如下所示:

```
MyStruct valType1 = new MyStruct();
IMyInterface refType = valType1;
```

然后使用一般的数据类型转换语法拆箱它:

```
MyStruct ValType2 = (MyStruct)refType;
```

从这些示例中可以看出,封箱是在没有用户干涉的情况下进行的(即不需要编写任何代码),但拆箱一个值需要进行显式转换,即需要进行数据类型转换(封箱是隐式的,所以不需要进行数据类型转换)。

读者可能想知道为什么要这么做。封箱非常有用,有两个非常重要的原因。首先,它允许在项的类型是 `object` 的集合(如 `ArrayList`)中使用值类型。其次,有一个内部机制允许在值类型上调用 `object`,例如 `int` 和结构。

最后需要注意的是,在访问值类型内容前,必须进行拆箱。

## 2. is 运算符

`is` 运算符并不是说明对象是某种类型的一种方式,而是可以检查对象是否是给定类型,或者是否可以转换为给定类型,如果是,这个运算符就返回 `true`。

在前面的示例中,有 `Cow` 和 `Chicken` 类,它们都继承于 `Animal`。使用 `is` 运算符比较 `Animal` 类型的对象,如果对象是这 3 种类型中的一种(不仅仅是 `Animal`),`is` 运算符就返回 `true`。使用前面介绍的 `GetType()` 方法和 `typeof()` 运算符很难做到这一点。

`is` 运算符的语法如下:

```
<operand> is <type>
```

这个表达式的结果如下:

- 如果 `<type>` 是一个类类型,而 `<operand>` 也是该类型,或者它继承了该类型,或者它可以封箱到该类型中,则结果为 `true`。
- 如果 `<type>` 是一个接口类型,而 `<operand>` 也是该类型,或者它是实现该接口的类型,则结果为 `true`。
- 如果 `<type>` 是一个值类型,而 `<operand>` 也是该类型,或者它可以拆箱到该类型中,则结果为 `true`。

下面用几个示例说明如何使用该运算符。

### 试一试：使用 is 运算符

- (1) 在 C:\BegVCSharp\Chapter11 目录中创建一个新控制台应用程序 Ch11Ex04。
- (2) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
namespace Ch11Ex04
{
    class Checker
    {
        public void Check(object param1)
        {
            if (param1 is ClassA)
                Console.WriteLine("Variable can be converted to ClassA.");
            else
                Console.WriteLine("Variable can't be converted to ClassA.");
            if (param1 is IMyInterface)
                Console.WriteLine("Variable can be converted to IMyInterface.");
            else
                Console.WriteLine("Variable can't be converted to IMyInterface.");

            if (param1 is MyStruct)
                Console.WriteLine("Variable can be converted to MyStruct.");
            else
                Console.WriteLine("Variable can't be converted to MyStruct.");
        }
    }

    interface IMyInterface
    {
    }

    class ClassA : IMyInterface
    {
    }

    class ClassB : IMyInterface
    {
    }
    class ClassC
    {
    }

    class ClassD : ClassA
    {
    }

    struct MyStruct : IMyInterface
    {
    }

    class Program
```

```

{
    static void Main(string[] args)
    {
        Checker check = new Checker();
        ClassA try1 = new ClassA();
        ClassB try2 = new ClassB();
        ClassC try3 = new ClassC();
        ClassD try4 = new ClassD();
        MyStruct try5 = new MyStruct();
        object try6 = try5;
        Console.WriteLine("Analyzing ClassA type variable:");
        check.Check(try1);
        Console.WriteLine("\nAnalyzing ClassB type variable:");
        check.Check(try2);
        Console.WriteLine("\nAnalyzing ClassC type variable:");
        check.Check(try3);
        Console.WriteLine("\nAnalyzing ClassD type variable:");
        check.Check(try4);
        Console.WriteLine("\nAnalyzing MyStruct type variable:");
        check.Check(try5);
        Console.WriteLine("\nAnalyzing boxed MyStruct type variable:");
        check.Check(try6);
        Console.ReadKey();
    }
}

```

代码段 Ch11Ex04\Program.cs

(3) 运行代码，其结果如图 11-6 所示。

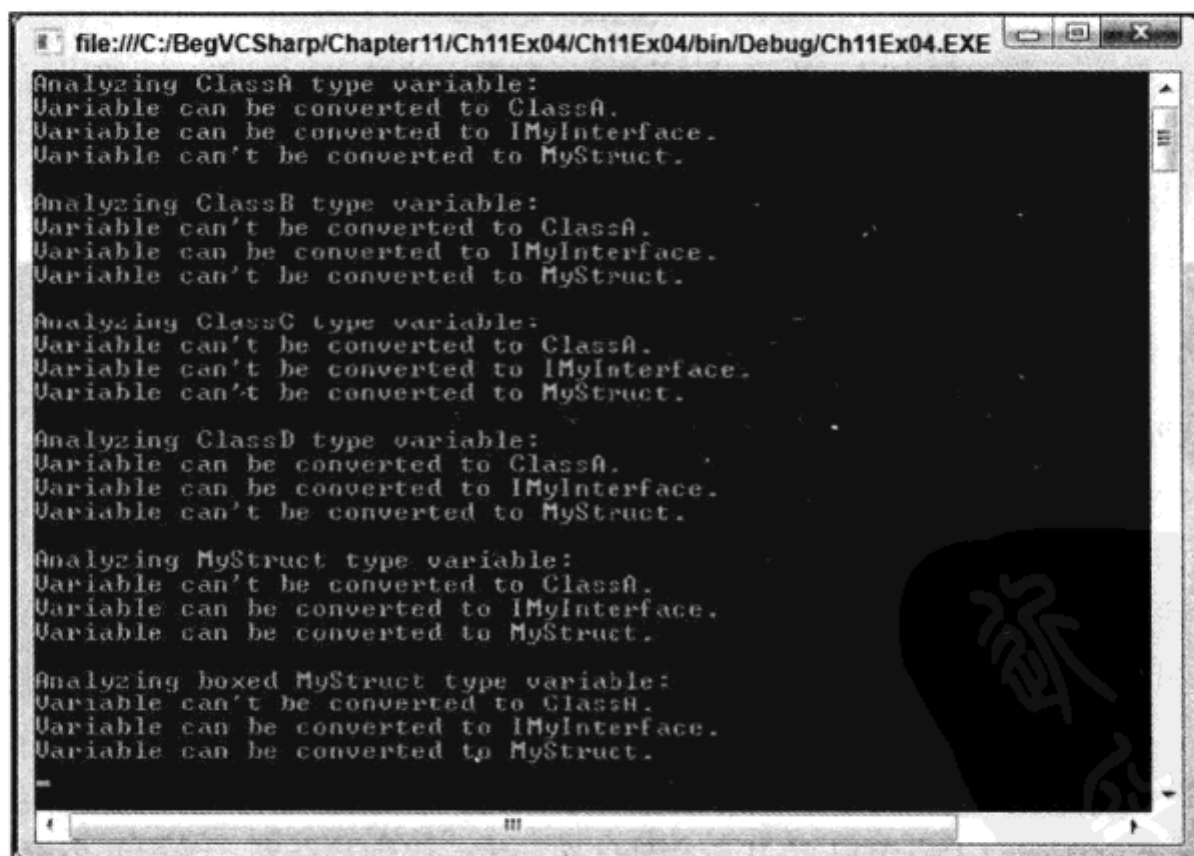


图 11-6

### 示例的说明

这个示例说明了使用 `is` 运算符的各种可能的结果。其中定义了 3 个类、一个接口和一个结构，并把它用作一个类的方法的参数，这个类使用 `is` 运算符确定它们是否可以转换为 `ClassA` 类型、接口类型和结构类型。

只有 `ClassA` 和 `ClassD`(继承了 `ClassA`)类型与 `ClassA` 兼容。如果一个类型没有继承一个类，该类型不会与该类兼容。

`ClassA`、`ClassB` 和 `MyStruct` 类型都实现了 `IMyInterface`，所以它们都与 `IMyInterface` 类型兼容。`ClassD` 继承了 `ClassA`，所以它们两个也兼容。因此，只有 `ClassC` 是不兼容的。

最后，只有 `MyStruct` 类型的变量本身和该类型的封箱变量与 `MyStruct` 兼容，因为不能把引用类型转换为值类型(当然，我们不能拆箱以前封箱的变量)。

### 11.2.2 值比较

考虑两个表示人的 `Person` 对象，它们都有一个 `Age` 整型属性。下面要比较它们，看看哪个人年龄较大。为此可以使用以下代码：

```
if (person1.Age > person2.Age)
{
    ...
}
```

这是可以的，还有其它方法，例如，使用下面的语法：

```
if (person1 > person2)
{
    ...
}
```

可以使用运算符重载，如本节后面所述。这是一个强大的技术，但应谨慎使用。在上面的代码中，年龄的比较不是非常明显，该段代码还可以比较身高、体重、IQ 等。

另一个方法是使用 `Comparable` 和 `Comparer` 接口，它们可以用标准的方式定义比较对象的过程。这是由 .NET Framework 中各种集合类提供的方式，是对集合中的对象进行排序的一种绝佳方式。

#### 1. 运算符重载

运算符重载(operator overloading)可以对我们设计的类使用标准的运算符，例如 `+`、`>` 等。这称为重载，因为在使用特定的参数类型时，我们为这些运算符提供了自己的实现代码，其方式与重载方法相同，也是为同名的方法提供不同的参数。

运算符重载非常有用，因为我们可以再运算符重载的实现中执行需要的任何操作，这并不像“把这两个操作数相加”这么简单。稍后介绍一个进一步升级 `CardLib` 库的示例。我们将提供比较运算符的实现代码，比较两张牌，看看在一圈(扑克牌游戏中的一局)中哪张牌会赢。

因为在许多扑克牌游戏中，一圈取决于牌的花色，这并不像比较牌上的数字那样直接。如果第二张牌与第一张牌的花色不同，则无论其点数是什么，第一张牌都会赢。考虑两个操作数的顺序，就可以实现这种比较。也可以考虑“王牌”的花色，而王牌可以胜过其他的花色，即使该王牌的花色与第一张牌不同，也是如此。也就是说，`card1 > card2` 是 `true`(这表示如果 `card1` 是第一个出牌，

则 card1 胜过了 card2), 并不意味着 `card2 > card1` 是 `false`。如果 card1 和 card2 都不是王牌, 且属于不同的花色, 则这两个比较都是 `true`。

但我们先看看运算符重载的基本语法。要重载运算符, 可给类添加运算符类型成员(它们必须是 `static`)。一些运算符有多种用途(如 `-` 运算符就有一元和二元两种功能), 因此我们还指定了要处理多少个操作数, 以及这些操作数的类型。一般情况下, 操作数的类型与定义运算符的类相同, 但也可以定义处理混合类型的运算符, 详见稍后的内容。

例如, 考虑一个简单类型 `AddClass1`, 如下所示:

```
public class AddClass1
{
    public int val;
}
```

这仅是 `int` 值的一个包装器(wrapper), 但可以用于说明原理。对于这个类, 下面的代码不能编译:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
AddClass1 op3 = op1 + op2;
```

其错误是 `+` 运算符不能应用于 `AddClass1` 类型的操作数, 因为我们尚未定义要执行的操作。下面的代码则可执行, 但得不到预期的结果:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
bool op3 = op1 == op2;
```

其中, 使用 `==` 二元运算符来比较 `op1` 和 `op2`, 看看它们是否引用同一个对象, 而不是验证它们的值是否相等。在上述代码中, 即使 `op1.val` 和 `op2.val` 相等, `op3` 也是 `false`。

要重载 `+` 运算符, 可使用下述代码:

```
public class AddClass1
{
    public int val;

    public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }
}
```

可以看出, 运算符重载看起来与标准静态方法声明类似, 但它们使用关键字 `operator` 和运算符本身, 而不是一个方法名。现在可以成功地使用 `+` 运算符和这个类, 如上面的示例所示:

```
AddClass1 op3 = op1 + op2;
```

重载所有的二元运算符都是一样的，一元运算符看起来也是类似的，但只有一个参数：

```
public class AddClass1
{
    public int val;

    public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }

    public static AddClass1 operator -(AddClass1 op1)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = -op1.val;
        return returnVal;
    }
}
```

这两个运算符处理的操作数的类型与类相同，返回值也是该类型，但考虑下面的类定义：

```
public class AddClass1
{
    public int val;

    public static AddClass3 operator +(AddClass1 op1, AddClass2 op2)
    {
        AddClass3 returnVal = new AddClass3();
        returnVal.val = op1.val + op2.val;
        return returnVal;
    }
}

public class AddClass2
{
    public int val;
}

public class AddClass3
{
    public int val;
}
```

下面的代码就可以执行：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;
```

可以酌情用这种方式混合类型。但要注意，如果把相同的运算符添加到 `AddClass2` 中，上面的代码就会失败，因为它弄不清要使用哪个运算符。因此，应注意不要把签名相同的运算符添加到多



个类中。

还要注意，如果混合了类型，操作数的顺序必须与运算符重载的参数顺序相同。如果使用了重载的运算符和顺序错误的操作数，操作就会失败。所以不能像下面这样使用运算符：

```
AddClass3 op3 = op2 + op1;
```

当然，除非提供了另一个重载运算符和倒序的参数：

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
    AddClass3 returnVal = new AddClass3();
    returnVal.val = op1.val + op2.val;
    return returnVal;
}
```

可以重载下述运算符：

- 一元运算符： +, -, !, ~, ++, --, true, false
- 二元运算符： +, -, \*, /, %, &, |, ^, <<, >>
- 比较运算符： ==, !=, <, >, <=, >=



如果重载 true 和 false 运算符，就可以在布尔表达式中使用类，例如，if(op1) {}。

不能重载赋值运算符，例如+=，但这些运算符使用与它们对应的简单运算符，例如+，所以不必担心它们。重载+表示+=像预期的那样执行。=运算符不能重载，因为它有一个基本的用途。但这个运算符与用户定义的转换运算符相关，详见下一节。

也不能重载&& 和 ||，但它们可以在计算中使用对应的运算符&和|，所以重载&和|就足够了。

一些运算符如< 和> 必须成对重载。这就是说，不能重载 <，除非也重载了>。在许多情况下，可以在这些运算符中调用其他运算符，以减少需要的代码数量(和可能发生的错误)，例如：

```
public class AddClass1
{
    public int val;

    public static bool operator >=(AddClass1 op1, AddClass1 op2)
    {
        return (op1.val >= op2.val);
    }

    public static bool operator <(AddClass1 op1, AddClass1 op2)
    {
        return !(op1 >= op2);
    }

    // Also need implementations for <= and > operators.
}
```

在较复杂的运算符定义中，这可以减少代码行数。这也意味着，如果后来决定修改这些运算符

的实现，需要改动的代码将较少。

这同样适用于 `==` 和 `!=`，但对于这些运算符，常常需要重写 `Object.Equals()` 和 `Object.GetHashCode()`，因为这两个函数也可以用于比较对象。重写这些方法，可以确保无论类的用户使用什么技术，都能得到相同的结果。这不太重要，但应增加进来，以保证其完整性。它需要下述非静态重写方法：

```
public class AddClass1
{
    public int val;

    public static bool operator ==(AddClass1 op1, AddClass1 op2)
    {
        return (op1.val == op2.val);
    }

    public static bool operator !=(AddClass1 op1, AddClass1 op2)
    {
        return !(op1 == op2);
    }

    public override bool Equals(object op1)
    {
        return val == ((AddClass1)op1).val;
    }

    public override int GetHashCode()
    {
        return val;
    }
}
```

`GetHashCode()` 可根据其状态，获取对象实例的一个唯一的 `int` 值。这里使用 `val` 就可以了，因为它也是一个 `int` 值。

注意，`Equals()` 使用 `object` 类型参数。我们需要使用这个签名，否则就将重载这个方法，而不是重写它。类的用户仍可以访问默认的实现代码。这样就必须使用数据类型转换得到需要的结果。这常常需要使用本章前面讨论的 `is` 运算符检查对象类型，代码如下所示：

```
public override bool Equals(object op1)
{
    if (op1 is AddClass1)
    {
        return val == ((AddClass1)op1).val;
    }
    else
    {
        throw new ArgumentException(
            "Cannot compare AddClass1 objects with objects of type "
            + op1.GetType().ToString());
    }
}
```


在这段代码中，如果传送给 `Equals` 的操作数的类型错误，或者不能转换为正确类型，就会抛出

一个异常。当然，这可能并不是我们希望的操作。我们要比较一个类型的对象和另一个类型的对象，此时需要更多的分支结构。另外，可能只允许对类型完全相同的两个对象进行比较，这需要对第一个 if 语句进行如下修改：

```
if (op1.GetType() == typeof(AddClass1))
```

## 2. 给 CardLib 添加运算符重载

现在再次升级 Ch11CardLib 项目，给 Card 类添加运算符重载。但首先给 Card 类添加额外的字段，指定某花色比其他花色大，使 A 有更高的级别。把这些字段指定为静态，因为设置了它们后，它们就可以应用到所有的 Card 对象上：

 可从 wrox.com 下载源代码

```
public class Card
{
    /// <summary>
    /// Flag for trump usage. If true, trumps are valued higher
    /// than cards of other suits.
    /// </summary>
    public static bool useTrumps = false;


    /// <summary>
    /// Trump suit to use if useTrumps is true.
    /// </summary>
    public static Suit trump = Suit.Club;

    /// <summary>
    /// Flag that determines whether aces are higher than kings or lower
    /// than deuces.
    /// </summary>
    public static bool isAceHigh = true;
}
```

代码段 Ch11CardLib\Card.cs

这些规则应用于应用程序中每个 Deck 的所有 Card 对象上。因此，两个 Deck 中的 Card 不可能遵守不同的规则。这适用于这个类库，但是确实可以做出这样的假设：如果一个应用程序要使用不同的规则，可以自行维护这些规则；例如，在切换牌时，设置 Card 的静态成员。

完成后，就要给 Deck 类再添加几个构造函数，以便用不同的特性初始化扑克牌：

 可从 wrox.com 下载源代码

```
/// <summary>
/// Nondefault constructor. Allows aces to be set high.
/// </summary>
public Deck(bool isAceHigh) : this()
{
    Card.isAceHigh = isAceHigh;
}

/// <summary>
/// Nondefault constructor. Allows a trump suit to be used.
/// </summary>
public Deck(bool useTrumps, Suit trump) : this()
```

```

{
    Card.useTrumps = useTrumps;
    Card.trump = trump;
}

/// <summary>
/// Nondefault constructor. Allows aces to be set high and a trump suit
/// to be used.
/// <summary>
public Deck(bool isAceHigh, bool useTrumps, Suit trump) : this()
{
    Card.isAceHigh = isAceHigh;
    Card.useTrumps = useTrumps;
    Card.trump = trump;
}

```

代码段 Ch11CardLib\Deck.cs

每个构造函数都使用第 9 章介绍的: this()语法来定义, 这样, 无论如何, 默认的构造函数总是在非默认的构造函数之前调用, 初始化扑克牌。

接着, 给 Card 类添加运算符重载(和推荐的重写代码):



可从  
WROX.COM  
下载源代码

```

public static bool operator ==(Card card1, Card card2)
{
    return (card1.suit == card2.suit) && (card1.rank == card2.rank);
}

public static bool operator !=(Card card1, Card card2)
{
    return !(card1 == card2);
}

public override bool Equals(object card)
{
    return this == (Card)card;
}

public override int GetHashCode()
{
    return 13*(int)rank + (int)suit;
}

public static bool operator >(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {
        if (isAceHigh)
        {
            if (card1.rank == Rank.Ace)
            {
                if (card2.rank == Rank.Ace)
                    return false;
                else
                    return true;
            }
        }
    }
}

```

```

        else
        {
            if (card2.rank == Rank.Ace)
                return false;
            else
                return (card1.rank > card2.rank);
        }
    }
    else
    {
        return (card1.rank > card2.rank);
    }
}
else
{
    if (useTrumps && (card2.suit == Card.trump))
        return false;
    else
        return true;
}
}

public static bool operator <(Card card1, Card card2)
{
    return !(card1 >= card2);
}

public static bool operator >=(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {
        if (isAceHigh)
        {
            if (card1.rank == Rank.Ace)
            {
                return true;
            }
            else
            {
                if (card2.rank == Rank.Ace)
                    return false;
                else
                    return (card1.rank >= card2.rank);
            }
        }
        else
        {
            return (card1.rank >= card2.rank);
        }
    }
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
    }
}

```

```

        else
            return true;
    }
}

public static bool operator <=(Card card1, Card card2)
{
    return !(card1 > card2);
}

```

代码段 Ch11CardLib\Card.cs

这段代码没有什么需要特别关注的，只是>和>=重载运算符的代码比较长。如果单步执行>运算符的代码，就可以看到它的执行情况，明白为什么需要这些步骤。

比较两张牌 card1 和 card2，其中 card1 假定为先出的牌。如前所述，在使用王牌时，这是很重要的，因为王牌胜过其他牌，即使非王牌比较大，也是这样。当然，如果两张牌的花色相同，则王牌是否也是该花色就不重要了，所以这是我们要进行的第一个比较：

```

public static bool operator >(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {

```

如果静态的 isAceHigh 标记为 true，就不能直接通过 Rank 枚举中的值比较牌的点数了。因为 A 的点数在这个枚举中是 1，比其他牌都小。此时就需要如下步骤：

- 如果第一张牌是 A，就检查第二张牌是否也是 A。如果是，则第一张牌就胜不过第二张牌。如果第二张牌不是 A，则第一张牌胜出：

```

        if (isAceHigh)
        {
            if (card1.rank == Rank.Ace)
            {
                if (card2.rank == Rank.Ace)
                    return false;
                else
                    return true;
            }

```

- 如果第一张牌不是 A，也需要检查第二张牌是否是 A。如果是，则第二张牌胜出；否则，就可以比较牌的点数，因为此时已不比较 A 了：

```

        else
        {
            if (card2.rank == Rank.Ace)
                return false;
            else
                return (card1.rank > card2.rank);
        }
    }
}

```

- 另外，如果 A 不是最大的，就只需比较牌的点数：

```

else
{
    return (card1.rank > card2.rank);
}

```

代码的其余部分主要考虑的是 card1 和 card2 花色不同的情况。其中静态 useTrumps 标记是非常重要的。如果这个标记是 true, 且 card2 是王牌, 则可以肯定, card1 不是王牌(因为这两张牌有不同的花色), 王牌总是胜出, 所以 card2 比较大:

```

else
{
    if (useTrumps && (card2.suit == Card.trump))
        return false;
}

```

如果 card2 不是王牌(或者 useTrumps 是 false), 则 card1 胜出, 因为它是最先出的牌:

```

else
    return true;
}
}

```

另有一个运算符(>=)使用与此类似的代码, 除此之外的其他运算符都非常简单, 所以不需要详细分析它们。

下面的简单客户代码测试这些运算符(把它放在客户项目的 Main() 函数中进行测试, 就像前面 CardLib 示例的客户代码那样):



可从  
wrox.com

下载源代码 Console.WriteLine("Clubs are trumps.");

```

Card card1, card2, card3, card4, card5;
card1 = new Card(Suit.Club, Rank.Five);
card2 = new Card(Suit.Club, Rank.Five);
card3 = new Card(Suit.Club, Rank.Ace);
card4 = new Card(Suit.Heart, Rank.Ten);
card5 = new Card(Suit.Diamond, Rank.Ace);

Console.WriteLine("{0} == {1} ? {2}",
    card1.ToString(), card2.ToString(), card1 == card2);
Console.WriteLine("{0} != {1} ? {2}",
    card1.ToString(), card3.ToString(), card1 != card3);
Console.WriteLine("{0}.Equals({1}) ? {2}",
    card1.ToString(), card4.ToString(), card1.Equals(card4));
Console.WriteLine("Card.Equals({0}, {1}) ? {2}",
    card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
Console.WriteLine("{0} > {1} ? {2}",
    card1.ToString(), card2.ToString(), card1 > card2);
Console.WriteLine("{0} <= {1} ? {2}",
    card1.ToString(), card3.ToString(), card1 <= card3);
Console.WriteLine("{0} > {1} ? {2}",
    card1.ToString(), card4.ToString(), card1 > card4);

```



```

Console.WriteLine("{0} > {1} ? {2}",
    card4.ToString(), card1.ToString(), card4 > card1);
Console.WriteLine("{0} > {1} ? {2}",
    card5.ToString(), card4.ToString(), card5 > card4);
Console.WriteLine("{0} > {1} ? {2}",
    card4.ToString(), card5.ToString(), card4 > card5);

Console.ReadKey();

```

代码段 Ch11CardClient\Program.cs

其结果如图 11-7 所示。

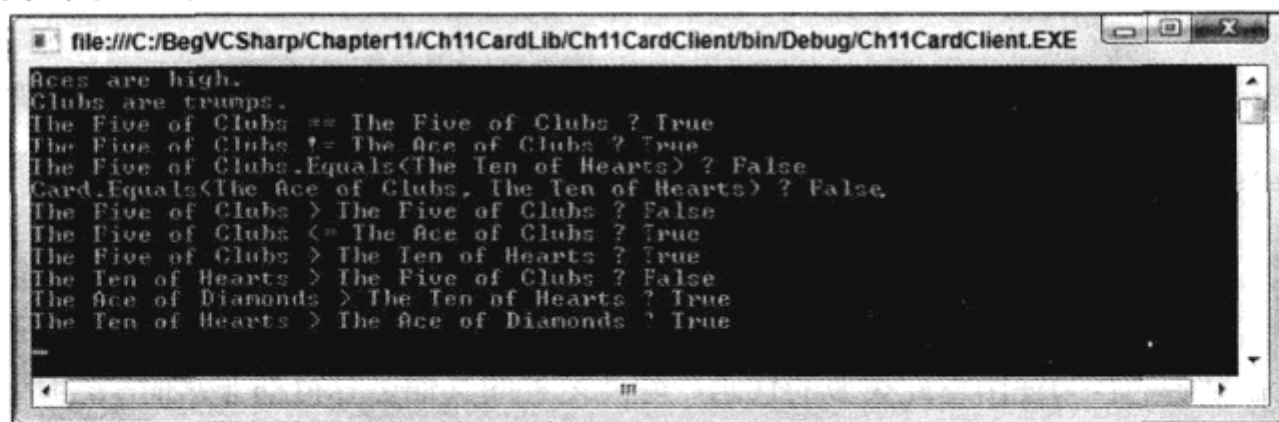


图 11-7

在两种情况下，在应用运算符时都考虑了指定的规则。这在输出结果的最后 4 行中尤其明显，说明王牌总是胜过其他牌。

### 3. IComparable 和 IComparer 接口

IComparable 和 IComparer 接口是 .NET Framework 中比较对象的标准方式。这两个接口之间的差别如下：

- IComparable 在要比较的对象的类中实现，可以比较该对象和另一个对象。
- IComparer 在一个单独的类中实现，可以比较任意两个对象。

一般使用 IComparable 给出类的默认比较代码，使用其他类给出非默认的比较代码。

IComparable 提供了一个方法 CompareTo()，这个方法接受一个对象。例如，实现可以为实现方法传送一个 Person 对象，以便确定这个人是否比当前的人更年老还是更年轻。实际上，这个方法返回一个 int，所以也可以确定第二个人与当前的人的年龄差：

```

if (person1.CompareTo(person2) == 0)
{
    Console.WriteLine("Same age");
}
else if (person1.CompareTo(person2) > 0)
{
    Console.WriteLine("person 1 is Older");
}
else
{
    Console.WriteLine("person1 is Younger");
}

```

`IComparer` 也提供了一个方法 `Compare()`。这个方法接受两个对象，返回一个整型结果，这与 `CompareTo()` 相同。对于支持 `IComparer` 的对象，可以使用下面的代码：

```
if (personComparer.Compare(person1, person2) == 0)
{
    Console.WriteLine("Same age");
}
else if (personComparer.Compare(person1, person2) > 0)
{
    Console.WriteLine("person 1 is Older");
}
else
{
    Console.WriteLine("person1 is Younger");
}
```

在这两种情况下，提供给方法的参数是 `System.Object` 类型。也就是说，可以比较其他任意类型的两个对象。所以，在返回结果之前，通常需要进行某种类型比较，如果使用了错误的类型，还会抛出异常。

.NET Framework 在类 `Comparer` 上提供了 `IComparer` 接口的默认实现方式，类 `Comparer` 位于 `System.Collections` 名称空间中，可以对简单类型以及支持 `IComparable` 接口的任意类型进行特定文化的比较。例如，可以通过下面的代码使用它：

```
string firstString = "First String";
string secondString = "Second String";
Console.WriteLine("Comparing '{0}' and '{1}', result: {2}",
    firstString, secondString,
    Comparer.Default.Compare(firstString, secondString));

int firstNumber = 35;
int secondNumber = 23;
Console.WriteLine("Comparing '{0}' and '{1}', result: {2}",
    firstNumber, secondNumber,
    Comparer.Default.Compare(firstNumber, secondNumber));
```

这里使用 `Comparer.Default` 静态成员获取 `Comparer` 类的一个实例，接着使用 `Compare()` 方法比较前两个字符串，之后比较两个整数，结果如下：

```
Comparing 'First String' and 'Second String', result: -1
Comparing '35' and '23', result: 1
```

在字母表中，F 在 S 的前面，所以 F “小于” S，第一个比较的结果就是 -1。同样，35 大于 23，所以结果是 1。注意这里的结果并未给出相差的幅度。

在使用 `Comparer` 时，必须使用可以比较的类型。例如，试图比较 `firstString` 和 `firstNumber` 就会生成一个异常。

下面是有关这个类的一些注意事项：

- 检查传送给 `Comparer.Compare()` 的对象，看看它们是否支持 `IComparable`。如果支持，就使用该实现代码。
- 允许使用 `null` 值，它表示“小于”其他对象。

- 字符串根据当前文化来处理。要根据不同的文化(或语言)处理字符串, `Comparer` 类必须使用其构造函数进行实例化, 以便传送指定所使用的文化的 `System.Globalization.CultureInfo` 对象。
- 字符串在处理时要区分大小写。如果要以不区分大小写的方式来处理它们, 就需要使用 `CaseInsensitiveComparer` 类, 该类以相同的方式工作。

#### 4. 使用 `Comparable` 和 `Comparer` 接口对集合排序

许多集合类可以用对象的默认比较方式进行排序, 或者用定制方法来排序。`ArrayList` 就是一个示例, 它包含方法 `Sort()`, 这个方法使用时可以不带参数, 此时使用默认的比较方式, 也可以给它传送 `Comparer` 接口, 以比较对象对。

在给 `ArrayList` 填充了简单类型时, 例如整数或字符串, 就会进行默认的比较。对于自己的类, 必须在类定义中实现 `Comparable`, 或者创建一个支持 `Comparer` 的类, 来进行比较。

注意, `System.Collection` 名称空间中的一些类, 包括 `CollectionBase`, 都没有提供排序方法。如果要对派生于这个类的集合排序, 就必须多做些工作, 自己给内部的 `List` 集合排序。

下面的示例说明如何使用默认的和非默认的比较方式给列表排序。

#### 试一试: 给列表排序

- (1) 在 `C:\BegVCSharp\Chapter11` 目录中创建一个新控制台应用程序 `Ch11Ex05`。
- (2) 添加一个新类 `Person`, 修改代码, 如下所示:



```
namespace Ch11Ex05
{
    class Person : IComparable
    {
        public string Name;
        public int Age;

        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }

        public int CompareTo(object obj)
        {
            if (obj is Person)
            {
                Person otherPerson = obj as Person;
                return this.Age - otherPerson.Age;
            }
            else
            {
                throw new ArgumentException(
                    "Object to compare to is not a Person object.");
            }
        }
    }
}
```

}

代码段 Ch11Ex05\Person.cs

(3) 添加一个新类 `PersonComparerName`, 修改代码, 如下所示:



```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex05
{
    public class PersonComparerName : IComparer
    {
        public static IComparer Default = new PersonComparerName();

        public int Compare(object x, object y)
        {
            if (x is Person && y is Person)
            {
                return Comparer.Default.Compare(
                    ((Person)x).Name, ((Person)y).Name);
            }
            else
            {
                throw new ArgumentException(
                    "One or both objects to compare are not Person objects.");
            }
        }
    }
}
```

代码段 Ch11Ex05\PersonComparerName.cs

(4) 修改 `Program.cs` 中的代码, 如下所示:



```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex05
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();
            list.Add(new Person("Jim", 30));
            list.Add(new Person("Bob", 25));
            list.Add(new Person("Bert", 27));
        }
    }
}
```

```

list.Add(new Person("Ernie", 22));

Console.WriteLine("Unsorted people:");
for (int i = 0; i < list.Count; i++)
{
    Console.WriteLine("{0} ({1})",
        (list[i] as Person).Name, (list[i] as Person).Age);
}
Console.WriteLine();

Console.WriteLine(
    "People sorted with default comparer (by age):");
list.Sort();
for (int i = 0; i < list.Count; i++)
{
    Console.WriteLine("{0} ({1})",
        (list[i] as Person).Name, (list[i] as Person).Age);
}
Console.WriteLine();

Console.WriteLine(
    "People sorted with nondefault comparer (by name):");
list.Sort(PersonComparerName.Default);
for (int i = 0; i < list.Count; i++)
{
    Console.WriteLine("{0} ({1})",
        (list[i] as Person).Name, (list[i] as Person).Age);
}

Console.ReadKey();
}
}
}

```

代码段 Ch11Ex05\Program.cs

(5) 执行代码，结果如图 11-8 所示。

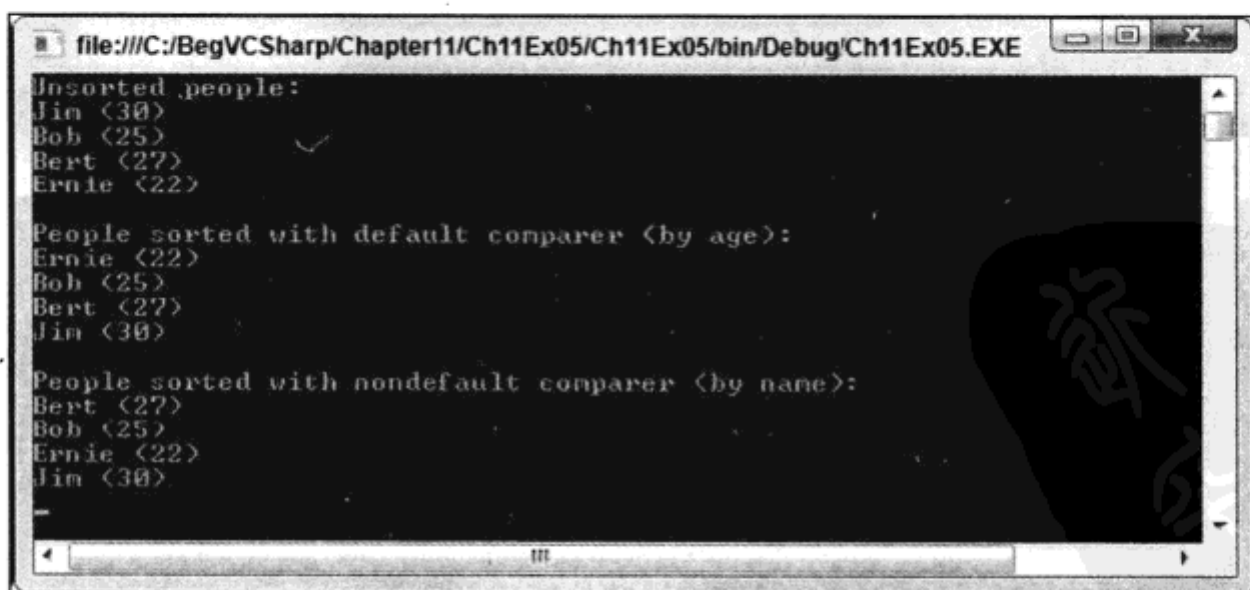


图 11-8

### 示例的说明

在这个示例中，包含 `Person` 对象的 `ArrayList` 用两种不同的方式排序。调用不带参数的 `ArrayList.Sort()` 方法，将使用默认的比较方式，即使用 `Person` 类中的 `CompareTo()` 方法(因为这个类实现了 `Comparable`)：

```
public int CompareTo(object obj)
{
    if (obj is Person)
    {
        Person otherPerson = obj as Person;
        return this.Age - otherPerson.Age;
    }
    else
    {
        throw new ArgumentException(
            "Object to compare to is not a Person object.");
    }
}
```

这个方法首先检查其参数是否能与 `Person` 对象比较，即该对象是否能转换为 `Person` 对象。如果遇到问题，就抛出一个异常。否则，就比较两个 `Person` 对象的 `Age` 属性。

接着，使用实现了 `IComparer` 的 `PersonComparerName` 类，执行非默认的比较排序。这个类有一个公共的静态字段，以方便使用：

```
public static IComparer Default = new PersonComparerName();
```

它可以使用 `PersonComparerName.Default` 获取一个实例，就像前面的 `Comparer` 类一样。这个类的 `CompareTo()` 方法如下：

```
public int Compare(object x, object y)
{
    if (x is Person && y is Person)
    {
        return Comparer.Default.Compare(
            ((Person)x).Name, ((Person)y).Name);
    }
    else
    {
        throw new ArgumentException(
            "One or both objects to compare are not Person objects.");
    }
}
```

这里也是首先检查参数，看看它们是否是 `Person` 对象，如果不是，就抛出一个异常；如果是，就使用默认的 `Comparer` 对象比较两个 `Person` 对象的字符串字段 `Name`。

## 11.3 转换

到目前为止，在需要把一种类型转换为另一种类型时，使用的都是类型转换。而这并不是唯一

的方式。在计算过程中, `int` 可以隐式转换为 `long` 或 `double`, 采用相同的方式还可以定义所创建的类型(隐式或显式)转换为其他类型的方式。为此, 可以重载转换运算符, 其方式与本章前面重载其他运算符的方式相同。本节第一部分就介绍重载方式。本节还将介绍另一个有用的运算符: `as` 运算符, 它一般适用于引用类型的转换。

### 11.3.1 重载转换运算符

除了重载如上所述的数学运算符之外, 还可以定义类型之间的隐式和显式转换。如果要在不相关的类型之间转换, 这是必须的, 例如, 如果在类型之间没有继承关系, 也没有共享接口, 这就是必须的。

下面定义 `ConvClass1` 和 `ConvClass2` 之间的隐式转换, 即编写下列代码:

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = op1;
```

另外, 还可以定义一个显式转换:

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = (ConvClass2)op1;
```

例如, 考虑下面的代码:

```
public class ConvClass1
{
    public int val;

    public static implicit operator ConvClass2(ConvClass1 op1)
    {
        ConvClass2 returnVal = new ConvClass2();
        returnVal.val = op1.val;
        return returnVal;
    }
}

public class ConvClass2
{
    public double val;

    public static explicit operator ConvClass1(ConvClass2 op1)
    {
        ConvClass1 returnVal = new ConvClass1();
        checked {returnVal.val = (int)op1.val;};
        return returnVal;
    }
}
```

其中, `ConvClass1` 包含一个 `int` 值, `ConvClass2` 包含一个 `double` 值。`int` 值可以隐式转换为 `double` 值, 所以可以在 `ConvClass1` 和 `ConvClass2` 之间定义一个隐式转换。但反过来就不行了, 应把 `ConvClass2` 和 `ConvClass1` 之间的转换定义为显式转换。

在代码中, 用关键字 `implicit` 和 `explicit` 来指定这些转换, 如上所示。对于这些类, 下面的代码就很好:



```
ConvClass1 op1 = new ConvClass1();
op1.val = 3;
ConvClass2 op2 = op1;
```

但反向的转换需要进行下述显式数据类型转换：

```
ConvClass2 op1 = new ConvClass2();
op1.val = 3e15;
ConvClass1 op2 = (ConvClass1)op1;
```

如果在显式转换中使用了 `checked` 关键字，则上述代码将产生一个异常，因为 `op1` 的 `val` 属性值太大，不能放在 `op2` 的 `val` 属性中。

### 11.3.2 as 运算符

`as` 运算符使用下面的语法，把一种类型转换为指定的引用类型：

```
<operand> as <type>
```

这只适用于下列情况：

- `<operand>` 的类型是 `<type>` 类型
- `<operand>` 可以隐式转换为 `<type>` 类型
- `<operand>` 可以封箱到 `<type>` 类型中

如果不能从 `<operand>` 转换为 `<type>`，则表达式的结果就是 `null`。

基类到派生类的转换可以使用显式转换来进行，但这并不总是有效的。考虑前面示例中的两个类 `ClassA` 和 `ClassD`，其中 `ClassD` 派生于 `ClassA`：

```
class ClassA : IMyInterface
{
}

class ClassD : ClassA
{
}
```

下面的代码使用 `as` 运算符把 `obj1` 中存储的 `ClassA` 实例转换为 `ClassD` 类型：

```
ClassA obj1 = new ClassA();
ClassD obj2 = obj1 as ClassD;
```

则 `obj2` 的结果为 `null`。

还可以使用多态性把 `ClassD` 实例存储在 `ClassA` 类型的变量中。下面的代码演示了这一点，`ClassA` 类型的变量包含 `ClassD` 类型的实例，使用 `as` 运算符把 `ClassA` 类型的变量转换为 `ClassD` 类型。

```
ClassD obj1 = new ClassD();
ClassA obj2 = obj1;
ClassD obj3 = obj2 as ClassD;
```

这次 `obj3` 最后包含与 `obj1` 相同的对象引用，而不是 `null`。

因此，`as` 运算符非常有用，因为下面使用简单类型转换的代码会抛出一个异常：

```
ClassA obj1 = new ClassA();
ClassD obj2 = (ClassD)obj1;
```

与此代码等价的 `as` 代码会把 `null` 值赋予 `obj2`，不会抛出异常。这表示，下面的代码(使用本章前面开发的两个类：`Animal` 和派生于 `Animal` 的一个类 `Cow`)在 C# 应用程序中是很常见的：

```
public void MilkCow(Animal myAnimal)
{
    Cow myCow = myAnimal as Cow;
    if (myCow != null)
    {
        myCow.Milk();
    }
    else
    {
        Console.WriteLine("{0} isn't a cow, and so can't be milked.",
            myAnimal.Name);
    }
}
```

这要比检查异常要简单得多！

## 11.4 小结

本章介绍的许多技巧都可以使 OOP 应用程序更强大、更有趣。尽管这些技巧要花一定的时间才能掌握，但它们可以使类更容易使用，简化了编写其他代码的任务。

本章介绍的每个论题都有许多用途。读者可能在几乎所有应用程序中见过某种形式的集合，如果要处理类型相同的一组对象，则创建类型安全的集合可以使任务更容易完成。介绍了集合后，我们又介绍了如何添加索引符和迭代器，以访问集合中的对象。

比较和转换是另一个很费时的领域。我们介绍了各种比较方式，以及封箱和拆箱的一些基本功能，还讨论了如何重载比较和转换运算符，如何利用列表排序把事物链接在一起。

第 12 章将介绍一个全新的内容：泛型，通过它们可以创建自动定制自身的类，动态地处理所选的类型。这对于集合来说非常有用，还会介绍如何使用泛型集合极大地简化本章的许多代码。

## 11.5 练习

(1) 创建一个集合类 `People`，它是下述 `Person` 类的集合，该集合中的项可以通过一个字符串索引符来访问，该字符串索引符是人名，与 `Person.Name` 属性相同：

```
public class Person
{
    private string name;
    private int age;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {

```

```
        name = value;
    }
}

public int Age
{
    get
    {
        return age;
    }
    set
    {
        age = value;
    }
}
```

- (2) 扩展上一题中的 Person 类，重载>、<、>=和<=运算符，比较 Person 实例的 Age 属性。
- (3) 给 People 类添加 GetOldest()方法，使用练习(2)中定义的重载运算符，返回其 Age 属性值为最大的 Person 对象数组(1 个或多个对象，因为对于这个属性而言，多个项可以有相同的值)。
- (4) 在 People 类上实现 ICloneable 接口，提供深度复制功能。
- (5) 给 People 类添加一个迭代器，在下面的 foreach 循环中获取所有成员的年龄：

```
foreach(int age in myPeople.Ages)
{
    //Display ages.
}
```

附录 A 给出了练习答案。

11.6 本章要点

主 题	重 要 概 念
定义集合	集合是可以包含其他类的实例的类。要定义集合，可以从 <code>CollectionBase</code> 中派生，或者自己实现集合接口，例如 <code>IEnumerable</code> 、 <code>ICollection</code> 和 <code> IList</code> 。一般需要为集合定义一个索引器，以使用 <code>collection.[index]</code> 语法来访问集合成员
字典	也可以定义关键字值集合，即字典，字典中的每一项都有一个关联的键。在字典中，键可以用于标识一项，而无需使用该项的索引。定义字典时，可以实现 <code>IDictionary</code> ，或者从 <code>DictionaryBase</code> 中派生类
迭代器	可以实现一个迭代器，来控制循环代码如何在其循环过程中获取值。要迭代一个类，需要实现 <code>GetEnumerator()</code> 方法，其返回类型是 <code>IEnumerator</code> 。要迭代类的成员，例如方法，可以使用 <code>IEnumerable</code> 返回类型。在迭代器的代码块中，使用 <code>yield</code> 关键字返回值
类型比较	使用 <code>GetType()</code> 方法可以获得对象的类型，使用 <code>typeof()</code> 运算符可以获得类的类型。可以比较这些类型值。还可以使用 <code>is</code> 运算符确定对象是否与某个类类型兼容
值比较	如果希望类的实例可以用标准的 C#运算符进行比较，就必须在类定义中重载这些运算符。对于其他类型的值比较，可以使用实现了 <code>IComparable</code> 或 <code>IComparer</code> 接口的类。这些接口特别适用于集合的排序
as 运算符	可以使用 <code>as</code> 运算符把一个值转换为引用类型。如果不能进行转换， <code>as</code> 运算符就返回 <code>null</code> 值



# 第 12 章

## 泛 型

### 本章内容:

---

- 泛型的含义
- 如何使用.NET Framework 提供的一些泛型类
- 如何定义自己的泛型
- 变体如何与泛型一起工作

C#第1版中最受诟病的一个方面是缺乏对泛型(generics)的支持。C++中的泛型(在该语言中称为模板)很早就被公认为是完成任务的最佳方式。它可以在编译期间由一个类型定义派生出许多特定的类型,这节省了大量的时间和精力。不知道是什么原因,泛型没有纳入C#的第1版,C#也因此受到很多批评。也许是因为泛型是一种很难掌握的技术,也许是开发人员觉得不需要泛型。幸好,C#2.0版中加入了泛型。泛型并不是真的很难掌握,只是需要用略微不同的方式处理而已。

本章首先介绍泛型的概念,先学习泛型的抽象术语,因为学习泛型的概念对高效使用它是至关重要的。

接着讨论.NET Framework中的一些泛型类型,这有助于更好地理解其功能和强大之处,以及在代码中需要使用的新语法。然后定义自己的泛型类型,包括泛型类、接口、方法和委托。还要介绍进一步定制泛型类型的其他技术:default关键字和类型约束。

最后讨论协变(covariance)和抗变(contravariance),这是C#4新增的两种形式的变体,在使用泛型类时提供了更大的灵活性。

### 12.1 泛型的概念

为了介绍泛型的概念,说明它们为什么这么有用,先回忆一下第11章中的集合类。基本集合可以包含在类似ArrayList这样的类中,但这些集合是没有类型化的,所以需要把object项转换为集合中实际存储的对象类型。继承自System.Object的任何对象都可以存储在ArrayList中,所以要特别仔细。假定包含在集合中的某些类型可能导致抛出异常,代码逻辑崩溃。前面介绍的技术可以处理

这个问题，包括检查对象类型所需的代码。

但是，更好的解决办法是一开始就使用强类型化的集合类。这种集合类派生于 `CollectionBase`，并可以拥有自己的方法，来添加、删除和访问集合的成员，但它可能把集合成员限制为派生于某种基本类型，或者必须支持某个接口。这会带来一个问题。每次创建需要包含在集合中的新类时，就必须执行下列任务之一：

- 使用某个集合类，该类已经定义为可以包含新类型的项。
- 创建一个新的集合类，它可以包含新类型的项，实现所有需要的方法。

一般情况下，新的类型需要额外的功能，我们常常并不需要新的集合类，创建集合类也会花费大量时间。

另一方面，泛型类大大简化了这个问题。泛型类是以实例化过程中提供的类型或类为基础建立的，可以毫不费力地对对象进行强类型化。对于集合，创建“T 类型对象的集合”十分简单，只需编写一行代码即可。不使用下面的代码：

```
CollectionClass items = new CollectionClass();
items.Add(new ItemClass());
```

而是使用：

```
CollectionClass<ItemClass> items = new CollectionClass<ItemClass>();
items.Add(new ItemClass());
```

尖括号语法就是把类型参数传送给泛型类型的方式。在上面的代码中，应把 `CollectionClass<ItemClass>` 看作 `ItemClass` 的 `CollectionClass`。当然，本章后面会详细探讨这个语法。

前面的泛型只涉及到集合，实际上泛型非常适合于这个领域，本章在后面介绍 `System.Collections.Generic` 名称空间时会提及。创建一个泛型类，就可以生成一些方法，它们的签名可以强类型化为我们需要的任何类型，该类型甚至可以是值类型或引用类型，处理各自的操作。还可以把用于实例化泛型类的类型限制为支持某个给定的接口，或派生自某种类型，只允许使用类型的一个子集。泛型并不限于类，还可以创建泛型接口、泛型方法(可以在非泛型类上定义)，甚至泛型委托。这将极大地提高代码的灵活性，正确使用泛型可以显著缩短开发时间。

那么该如何实现泛型呢？通常，在创建类时，它会编译为一个类型，然后在代码中使用。读者可能认为，在创建泛型类时，它必须被编译为许多类型，才能进行实例化。幸好并不是这样：在 .NET 中，类有无限多个。在后台，.NET 运行库允许在需要时动态生成泛型类。在通过实例化来请求生成之前，B 的某个泛型类 A 甚至不存在。



对于熟悉 C++ 或者对 C++ 感兴趣的读者来说，这是 C++ 模板和 C# 泛型类的一个区别。在 C++ 中，编译器可以检测出在哪里使用了模板的某个特定类型，例如，模板 B 的 A 类型，然后编译需要的代码，来创建这个类型。而在 C# 中，所有操作都在运行期间进行。

总之，泛型允许灵活地创建类型，处理一种或多种特定类型的对象，这些类型是在实例化时确定的，否则就使用泛型类型。下面看看如何在实际中使用它们。

## 12.2 使用泛型

在探讨如何创建自己的泛型类型之前，先介绍.NET Framework 提供的泛型，包括 `System.Collections.Generic` 名称空间中的类型，这个名称空间已在前面的代码中出现过多次，因为默认情况下它包含在控制台应用程序中。我们还没有使用过这个名称空间中的类型，但下面就要使用了。本节将讨论这个名称空间中的类型，以及如何使用它们创建强类型化的集合，改进已有集合的功能。

首先论述另一个较简单的泛型类型，即可空类型(nullable type)，它解决了值类型的一个小问题。

### 12.2.1 可空类型

在前面的章节中，介绍了值类型(大多数基本类型，例如，`int`、`double`和所有的结构)区别于引用类型(`string`和所有的类)的一种方式：值类型必须包含一个值，它们可以在声明之后、赋值之前，在未赋值的状态下存在，但不能以任何方式使用。而引用类型可以是 `null`。

有时让值类型为空是很有用的(尤其是处理数据库时)，泛型使用 `System.Nullable<T>` 类型提供了使值类型为空的一种方式。例如：

```
System.Nullable<int> nullableInt;
```

这行代码声明了一个变量 `nullableInt`，它可以拥有 `int` 变量能包含的任意值，还可以拥有值 `null`。所以可以编写如下的代码：

```
nullableInt = null;
```

如果 `nullableInt` 是一个 `int` 类型的变量，上面的代码是不能编译的。

前面的赋值等价于：

```
nullableInt = new System.Nullable<int>();
```

与其他任何变量一样，无论是初始化为 `null`(使用上面的语法)，还是通过给它赋值来初始化，都不能在初始化之前使用它。

可以像测试引用类型一样，测试可空类型，看看它们是否为 `null`：

```
if (nullableInt == null)
{
    ...
}
```

另外，可以使用 `HasValue` 属性：

```
if (nullableInt.HasValue)
{
    ...
}
```

这不适用于引用类型，即使引用类型有一个 `HasValue` 属性，也不能使用这种方法，因为引用类型的变量值为 `null`，就表示不存在对象，当然就不能通过对象来访问这个属性，否则会抛出一个



异常。

使用 `Value` 属性可以查看可空类型的值。如果 `HasValue` 是 `true`，就说明 `Value` 属性有一个非空值。但如果 `HasValue` 是 `false`，就说明变量被赋予了 `null`，访问 `Value` 属性会抛出 `System.InvalidOperationException` 类型的异常。

可空类型非常有用，以致于修改了 C# 语法。声明可空类型的变量不使用上述语法，而是使用下面的语法：

```
int? nullableInt;
```

`int?` 是 `System.Nullable<int>` 的缩写，但更便于读取。在后面的章节中就使用了这个语法。

### 1. 运算符和可空类型

对于简单类型如 `int`，可以使用 `+`、`-` 等运算符来处理值。而对于可空类型，这是没有区别的：包含在可空类型中的值会隐式转换为需要的类型，使用适当的运算符。这也适用于结构和自己提供的运算符。例如：

```
int? op1 = 5;
int? result = op1 * 2;
```

注意，其中 `result` 变量的类型也是 `int?`。下面的代码不会被编译：

```
int? op1 = 5;
int result = op1 * 2;
```

为了使上面的代码正常工作，必须进行显式转换：

```
int? op1 = 5;
int result = (int)op1 * 2;
```

或者通过 `Value` 属性访问值，需要的代码如下：

```
int? op1 = 5;
int result = op1.Value * 2;
```

只要 `op1` 有一个值，上面的代码就可以正常运行。如果 `op1` 是 `null`，就会生成 `System.InvalidOperationException` 类型的异常。

这就引出了下一个问题：当运算等式中的一个或两个值是 `null` 时，例如，下面代码中的 `op1`，会发生什么情况？

```
int? op1 = null;
int? op2 = 5;
int? result = op1 * op2;
```

答案是：对于除了 `bool?` 之外的所有简单可空类型，该操作的结果是 `null`，可以把它解释为“不能计算”。对于结构，可以定义自己的运算符来处理这种情况(详见本章后面的内容)。对于 `bool?`，为 `&` 和 `|` 定义的运算符会得到非空返回值，如表 12-1 所示。

表 12-1

op1	op2	op1 & op2	op1   op2
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

这些运算符的结果十分符合逻辑，如果不需要知道其中一个操作数的值，就可以计算出结果，则该操作数是否为 null 就不重要。

2. ??运算符

为了进一步减少处理可空类型所需的代码量，使可空变量的处理变得更简单，可以使用??运算符。这个运算符称为空接合运算符(null coalescing operator)，是一个二元运算符，允许给可能等于 null 的表达式提供另一个值。如果第一个操作数不是 null，该运算符就等于第一个操作数，否则，该运算符就等于第二个操作数。下面的两个表达式的作用是相同的：

```
op1 ?? op2
op1 == null ? op2 : op1
```

在这两行代码中，op1 可以是任意可空表达式，包括引用类型和更重要的可空类型。因此，如果可空类型是 null，就可以使用??运算符提供要使用的默认值，如下所示：

```
int? op1 = null;
int result = op1 * 2 ?? 5;
```

在这个示例中，op1 是 null，所以 op1\*2 也是 null。但是，??运算符检测到这个情况，并把值 5 赋予 result。这里要特别注意，在结果中放入 int 类型的变量 result 不需要显式转换。??运算符会自动处理这个转换。还可以把??等式的结果放在 int?中：

```
int? result = op1 * 2 ?? 5;
```

在处理可空变量时，??运算符有许多用途，它也是一种提供默认值的便捷方式，不需要使用 if 结构中的代码块或容易引起混淆的三元运算符。

在下面的示例中，将介绍可空类型 Vector。

试一试：可空类型

- (1) 在 C:\BegVCSharp\Chapter12 目录中创建一个新控制台应用程序项目 Ch12Ex01。
- (2) 在文件 Vector.cs 中添加一个新类 Vector。
- (3) 修改 Vector.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
public class Vector
{
    public double? R = null;
    public double? Theta = null;

    public double? ThetaRadians
    {
        get
        {
            // Convert degrees to radians.
            return (Theta * Math.PI / 180.0);
        }
    }

    public Vector(double? r, double? theta)
    {
        // Normalize.
        if (r < 0)
        {
            r = -r;
            theta += 180;
        }
        theta = theta % 360;

        // Assign fields.
        R = r;
        Theta = theta;
    }

    public static Vector operator +(Vector op1, Vector op2)
    {
        try
        {
            // Get (x, y) coordinates for new vector.
            double newX = op1.R.Value * Math.Sin(op1.ThetaRadians.Value)
                + op2.R.Value * Math.Sin(op2.ThetaRadians.Value);
            double newY = op1.R.Value * Math.Cos(op1.ThetaRadians.Value)
                + op2.R.Value * Math.Cos(op2.ThetaRadians.Value);

            // Convert to (r, theta).
            double newR = Math.Sqrt(newX * newX + newY * newY);
            double newTheta = Math.Atan2(newX, newY) * 180.0 / Math.PI;

            // Return result.
            return new Vector(newR, newTheta);
        }
        catch
        {
            // Return "null" vector.
            return new Vector(null, null);
        }
    }

    public static Vector operator -(Vector op1)
    {

```

```

        return new Vector(-op1.R, op1.Theta);
    }

    public static Vector operator -(Vector op1, Vector op2)
    {
        return op1 + (-op2);
    }

    public override string ToString()
    {
        // Get string representation of coordinates.
        string rString = R.HasValue ? R.ToString() : "null";
        string thetaString = Theta.HasValue ? Theta.ToString() : "null";

        // Return (r, theta) string.
        return string.Format("{0}, {1}", rString, thetaString);
    }
}

```

代码段 Ch12Ex01\Vector.cs

(4) 修改 Program.cs 中的代码, 如下所示:



```

class Program
{
    static void Main(string[] args)
    {
        Vector v1 = GetVector("vector1");
        Vector v2 = GetVector("vector1");
        Console.WriteLine("{0} + {1} = {2}", v1, v2, v1 + v2);
        Console.WriteLine("{0} - {1} = {2}", v1, v2, v1 - v2);
        Console.ReadKey();
    }

    static Vector GetVector(string name)
    {
        Console.WriteLine("Input {0} magnitude:", name);
        double? r = GetNullableDouble();
        Console.WriteLine("Input {0} angle (in degrees):", name);
        double? theta = GetNullableDouble();
        return new Vector(r, theta);
    }

    static double? GetNullableDouble()
    {
        double? result;
        string userInput = Console.ReadLine();
        try
        {
            result = double.Parse(userInput);
        }
        catch
        {
            result = null;
        }
        return result;
    }
}

```

```
}  
}
```

代码段 Ch12Ex01\Program.cs

(5) 执行应用程序，给两个矢量(vector)输入值，示例输出结果如图 12-1 所示。



图 12-1

(6) 再次执行应用程序，这次跳过四个值中的至少一个，示例输出结果如图 12-2 所示。

示例的说明

在这个示例中，创建了一个类Vector，它表示带极坐标(有一个幅值和一个角度)的矢量，如图 12-3 所示。



图 12-2

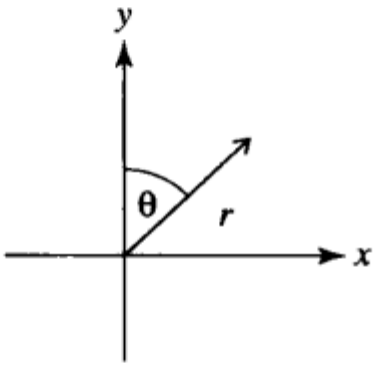


图 12-3

坐标 r 和 θ 在代码中用公共字段 R 和 Theta 表示，其中 Theta 的单位是度(°)。ThetaRadians 用于获取 Theta 的弧度值，这是必需的，因为 Math 类在其静态方法中使用弧度。R 和 Theta 的类型都是 double?，所以它们可以为空。



```
public class Vector  
{  
    public double? R = null;  
    public double? Theta = null;  
  
    public double? ThetaRadians  
    {  
        get  
        {  
            // Convert degrees to radians.  
            return (Theta * Math.PI / 180.0);  
        }  
    }  
}
```

代码段 Ch12Ex01\Vector.cs

Vector 的构造函数标准化 R 和 Theta 的初始值，然后赋予公共字段。

```
public Vector(double? r, double? theta)
{
    // Normalize.
    if (r < 0)
    {
        r = -r;
        theta += 180;
    }
    theta = theta % 360;

    // Assign fields.
    R = r;
    Theta = theta;
}
```

Vector 类的主要功能是使用运算符重载对矢量进行相加和相减运算，这需要一些非常基本的三角函数知识，这里不解释它们。在代码中，重要的是，如果在获取 R 或 ThetaRadians 的 Value 属性时抛出了异常，即其中一个是 null，就返回“空”矢量。

```
public static Vector operator +(Vector op1, Vector op2)
{
    try
    {
        // Get (x, y) coordinates for new vector.
        ...
    }
    catch
    {
        // Return "null" vector.
        return new Vector(null, null);
    }
}
```

如果组成矢量的一个坐标是 null，该矢量就是无效的，这里用 R 和 Theta 都可为 null 的 Vector 类来表示。Vector 类的其他代码重写了其他运算符，以便扩展相加的功能，使之包含相减操作，再重写 ToString()，获取 Vector 对象的字符串表示。

Program.cs 中的代码测试 Vector 类，让用户初始化两个矢量，再对它们进行相加和相减。如果用户省略了某个值，该值就解释为 null，应用前面提及的规则。

### 12.2.2 System.Collections.Generic 名称空间

实际上，本书前面的每个应用程序都包含如下名称空间：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

System 名称空间包含 .NET 应用程序使用的大多数基本类型。System.Text 名称空间包含与字符串处理和编码相关的类型，System.Linq 名称空间将从第 23 章开始介绍。但 System.Collections.Generic 名称空间包含什么类型？为什么要在默认情况下把它包含在控制台应用程序中？

这个名称空间包含用于处理集合的泛型类型，使用得非常频繁，用 using 语句配置它，使用时就不必添加限定符了。

本章前面提到过这些泛型类型，下面将予以介绍，它们可以使工作更容易完成，可以毫不费力地创建强类型化的集合类。表 12-2 描述了本节要介绍的 System.Collections.Generic 名称空间中的两个类型，本章后面还会详细阐述这个名称空间中的更多类型。

表 12-2

类 型	说 明
List<T>	T 类型对象的集合
Dictionary<K, V>	与 K 类型的键值相关的 V 类型的项的集合

后面还会介绍和这些类一起使用的各种接口和委托。

1. List<T>

List<T> 泛型集合类型更加快捷、更易于使用；这样，就不必像上一章那样，从 CollectionBase 中派生一个类，然后实现需要的方法。它的另一个好处是正常情况下需要实现的许多方法(例如，Add())已经自动实现了。

创建 T 类型对象的集合需要如下代码：

```
List<T> myCollection = new List<T>();
```

这就足够了。没有定义类、实现方法和进行其他操作。还可以把 List<T> 对象传送给构造函数，在集合中设置项的起始列表。使用这个语法实例化的对象将支持表 12-3 中的方法和属性(其中，提供给 List<T> 泛型的类型是 T)。

表 12-3

成 员	说 明
int Count	该属性给出集合中项的个数
void Add(T item)	把一个项添加到集合中
void AddRange(IEnumerable<T>)	把多个项添加到集合中
ICollection<T> AsReadOnly()	给集合返回一个只读接口
int Capacity	获取或设置集合可以包含的项数
void Clear()	删除集合中的所有项
bool Contains(T item)	确定 item 是否包含在集合中
void CopyTo(T[] array, int index)	把集合中的项复制到数组 array 中，从数组的索引 index 开始
IEnumerator<T> GetEnumerator()	获取一个 IEnumerator<T> 实例，用于迭代集合。注意，返回的接口强类型化为 T，所以在 foreach 循环中不需要类型转换



(续表)

成 员	说 明
int IndexOf(T item)	获取 item 的索引，如果集合中并未包含该项，就返回 - 1
void Insert(int index, T item)	把 item 插入到集合的指定索引位置上
bool Remove(T item)	从集合中删除第一个 item，并返回 true；如果 item 不包含在集合中，就返回 false
void RemoveAt(int index)	从集合中删除索引 index 处的项


List<T>还有一个 Item 属性，允许进行类似于数组的访问，如下所示：

```
T itemAtIndex2 = myCollectionOfT[2];
```

这个类还支持其他几个方法，但只要掌握了上述知识，就完全可以开始使用该类了。下面的示例将介绍如何在实际中使用 List<T>。

试一试：使用 List<T>

- (1) 在 C:\BegVCSharp\Chapter12 目录中创建一个新控制台应用程序 Ch12Ex02。
- (2) 在 Solution Explorer 窗口中右击项目名称，选择 Add | Existing Item 选项。
- (3) 在 C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01 目录中选择 Animal.cs、Cow.cs 和 Chicken.cs 文件，单击 Add 按钮。
- (4) 修改这 3 个文件中的名称空间声明，如下所示：




可从  
wrox.com  
下载源代码

```
namespace Ch12Ex02
```

---

Code snippets Ch12Ex02\Animal.cs、Ch12Ex02\Cow.cs 和 Ch12Ex02\Chicken.cs

- (5) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    List<Animal> animalCollection = new List<Animal>();
    animalCollection.Add(new Cow("Jack"));
    animalCollection.Add(new Chicken("Vera"));
    foreach (Animal myAnimal in animalCollection)
    {
        myAnimal.Feed();
    }
    Console.ReadKey();
}
```

---

代码段 Ch12Ex02\Program.cs

- (6) 执行应用程序，结果与第 11 章的 Ch11Ex02 的结果相同。

示例的说明

这个示例与 Ch11Ex02 只有两个区别。第一个区别是下面的代码：

```
Animals animalCollection = new Animals();
```

被替换为：

```
List<Animal> animalCollection = new List<Animal>();
```


第二个区别比较重要：项目中不再有 Animals 集合类。通过使用泛型的集合类，前面为创建这个类所做的工作现在用一行代码即可完成。

获得相同效果的另一个方法是不修改 Program.cs 中的代码，而是使用 Animals 的如下定义：

```
public class Animals : List<Animal>
{
}
```

这么做的优点是，能较容易地看懂 Program.cs 中的代码，还可以在合适时给 Animals 类添加额外的成员。

为什么不从CollectionBase中派生类？这是一个很好的问题。实际上，在许多情况下，我们都不会从CollectionBase中派生类。知道内部工作原理肯定是件好事，因为 List<T>以相同的方式工作，但CollectionBase主要用于向后兼容。使用CollectionBase的唯一场合是要更多地控制向类的用户展示的成员。例如，如果希望集合类的 Add()方法使用内部访问修饰符，则使用 CollectionBase 是最佳选择。



也可以把要使用的初始容量(作为 int)传递给 List<T>的构造函数，或者传递给使用 IEnumerable<T>接口的初始项列表。支持这个接口的类包括 List<T>。

2. 对泛型列表进行排序和搜索

对泛型列表进行排序与对其他列表进行排序是一样的。第 11 章介绍了如何使用 IComparer 和 IComparable 接口比较两个对象，然后对该类型的对象列表排序。这里唯一的区别是，可以使用泛型接口 IComparer<T>和 IComparable<T>，它们提供了略有区别、且针对特定类型的方法。表 12-4 列出了它们之间的区别。

表 12-4

泛 型 方 法	非泛型方法	区 别
int IComparable<T>. CompareTo(T otherObj)	int IComparable. CompareTo(object, otherObj)	在泛型版本中是强类型化的
bool IComparable<T>. Equals(T otherObj)	N/A	在非泛型接口中不存在，可以使用 object.Equals()替代
int IComparer<T>. Compare(TobjectA,TobjectB)	int IComparer. Compare(objectobjectA,objectobjectB)	、在泛型版本中是强类型化的

(续表)

泛 型 方 法	非泛型方法	区 别
<code>bool IComparer&lt;T&gt;.Equals(T objectA, T objectB)</code>	N/A	在非泛型接口中不存在，可以改用 <code>object.Equals()</code>
<code>int IComparer&lt;T&gt;.GetHashCode (T objectA)</code>	N/A	在非泛型接口中不存在，可以改用继承的 <code>object.GetHashCode ()</code>

要对 `List<T>` 排序，可以在要排序的类型上提供 `IComparable<T>` 接口，或者提供 `Comparer<T>` 接口。另外，还可以提供泛型委托，作为排序方法。从了解工作原理的角度来看，这非常有趣，因为实现上述接口并不比实现其非泛型版本更麻烦。

一般情况下，给列表排序需要有一个方法来比较两个 `T` 类型的对象。要在列表中搜索，也需要一个方法来检查 `T` 类型的对象，看看它是否满足某个条件。定义这样的方法很简单，这里给出两个可以使用的泛型委托类型：

- `Comparison<T>`：这个委托类型用于排序方法，其返回类型和参数如下：

```
int method (T objectA, T objectB)
```

- `Predicate<T>`：这个委托类型用于搜索方法，其返回类型和参数如下：

```
bool method(T targetObject)
```

可以定义任意多个这样的方法，使用它们实现 `List<T>` 的搜索和排序方法。下面的示例进行了演示。

试一试：List<T>的搜索和排序

- (1) 在 `C:\BegVCSharp\Chapter12` 目录中创建一个新控制台应用程序 `Ch12Ex03`。
- (2) 在 `Solution Explorer` 窗口中右击项目名称，选择 `Add | Add Existing Item` 选项。
- (3) 在 `C:\BegVCSharp\Chapter12\Ch12Ex01\Ch12Ex01` 目录中选择 `Vector.cs` 文件，单击 `Add` 按钮。
- (4) 修改这个文件中的名称空间声明，如下所示：

```
namespace Ch12Ex03
```

- (5) 添加一个新类 `Vectors`。
- (6) 修改 `Vectors.cs` 中的代码，如下所示：



```
public class Vectors : List<Vector>
{
    public Vectors()
    {
    }

    public Vectors(IEnumerable<Vector> initialItems)
    {
        foreach (Vector vector in initialItems)
```



```

        {
            Add(vector);
        }
    }

    public string Sum()
    {
        StringBuilder sb = new StringBuilder();
        Vector currentPoint = new Vector(0.0, 0.0);
        sb.Append("origin");
        foreach (Vector vector in this)
        {
            sb.AppendFormat(" + {0}", vector);
            currentPoint += vector;
        }
        sb.AppendFormat(" = {0}", currentPoint);
        return sb.ToString();
    }
}

```

代码段 Ch12Ex03\Vector.cs

- (7) 添加一个新类 VectorDelegates。
- (8) 修改 VectorDelegates.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

public static class VectorDelegates
{
    public static int Compare(Vector x, Vector y)
    {
        if (x.R > y.R)
        {
            return 1;
        }
        else if (x.R < y.R)
        {
            return -1;
        }
        return 0;
    }

    public static bool TopRightQuadrant(Vector target)
    {
        if (target.Theta >= 0.0 && target.Theta <= 90.0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

代码段 Ch12Ex03\VectorDelegates.cs

(9) 修改 Program.cs 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    Vectors route = new Vectors();
    route.Add(new Vector(2.0, 90.0));
    route.Add(new Vector(1.0, 180.0));
    route.Add(new Vector(0.5, 45.0));
    route.Add(new Vector(2.5, 315.0));

    Console.WriteLine(route.Sum());

    Comparison<Vector> sorter = new Comparison<Vector>
        (VectorDelegates.Compare);
    route.Sort(sorter);
    Console.WriteLine(route.Sum());

    Predicate<Vector> searcher =
        new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
    Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));
    Console.WriteLine(topRightQuadrantRoute.Sum());

    Console.ReadKey();
}
```

代码段 Ch12Ex03\Program.cs

(10) 执行应用程序, 结果如图 12-4 所示。

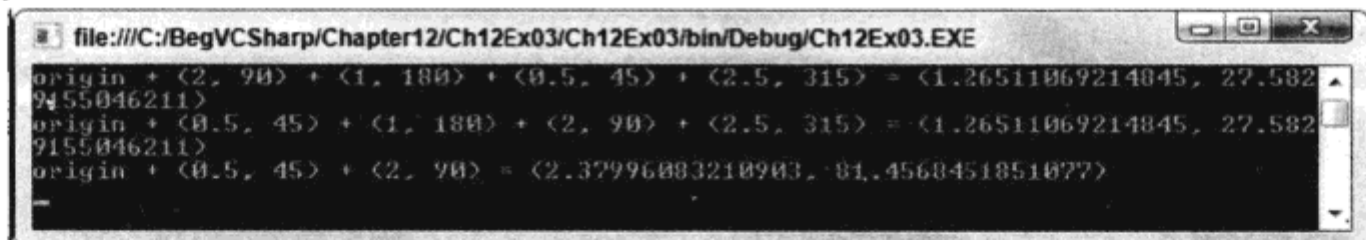


图 12-4

#### 示例的说明

在这个示例中, 为 Ch12Ex01 中的 Vector 类创建了一个集合类 Vectors。可以只使用 List<Vector> 类型的变量, 但因为需要其他功能, 所以使用了一个新类 Vectors, 它派生自 List<Vector>, 允许添加需要的其他成员。

该类有一个成员 Sum(), 依次返回每个矢量的字符串列表, 并在最后把它们加在一起(使用源类 Vector 的重载+运算符)。每个矢量都可以看作“方向+距离”, 所以这个矢量列表构成了一条有端点的路径。



可从  
wrox.com  
下载源代码

```
public string Sum()
{
    StringBuilder sb = new StringBuilder();
    Vector currentPoint = new Vector(0.0, 0.0);
    sb.Append("origin");
    foreach (Vector vector in this)
    {
        sb.AppendFormat(" + {0}", vector);
    }
}
```

```

        currentPoint += vector;
    }
    sb.AppendFormat(" = {0}", currentPoint);
    return sb.ToString();
}

```

代码段 Ch12Ex03\Vector.cs

这个方法使用 `System.Text` 名称空间中的简便的 `StringBuilder` 类来构建响应字符串。这个类包含 `Append()` 和 `AppendFormat()` 等成员(这里使用), 所以很容易组合字符串, 其性能也比串联各个字符串要高。使用这个类的 `ToString()` 方法即可获得最终的字符串。

本例还创建了两个用作委托的方法, 作为 `VectorDelegates` 的静态成员。`Compare()` 用于比较(排序), `TopRightQuadrant()` 用于搜索。稍后在讨论 `Program.cs` 中的代码时介绍它们。

`Main()` 中的代码首先初始化 `Vectors` 集合, 给它添加几个 `Vector` 对象:



可从  
wrox.com  
下载源代码

```

Vectors route = new Vectors();
route.Add(new Vector(2.0, 90.0));
route.Add(new Vector(1.0, 180.0));
route.Add(new Vector(0.5, 45.0));
route.Add(new Vector(2.5, 315.0));

```

代码段 Ch12Ex03\Program.cs

如前所述, `Vectors.Sum()` 方法用于输出集合中的项, 这次是按照其初始顺序输出:

```
Console.WriteLine(route.Sum());
```

接着, 创建第一个委托 `sorter`, 这个委托属于 `Comparison<Vector>` 类型, 因此可以赋予带如下返回类型和参数的方法:

```
int method(Vector objectA, Vector objectB)
```

它匹配 `VectorDelegates.Compare()`, 该方法就是赋予委托的方法。

```
Comparison<Vector> sorter = new Comparison<Vector>
    (VectorDelegates.Compare);
```

`Compare()` 比较两个矢量的大小, 如下所示:

```

public static int Compare(Vector x, Vector y)
{
    if (x.R > y.R)
    {
        return 1;
    }
    else if (x.R < y.R)
    {
        return -1;
    }
    return 0;
}

```

这样就可以按大小对矢量排序了：

```
route.Sort(sorter);
Console.WriteLine(route.Sum());
```

应用程序的输出结果符合我们的预期——汇总的结果是一样的，因为无论用什么顺序执行各个步骤，“矢量路径”的端点都是相同的。

然后，进行搜索，获取集合中的一个矢量子集。这需要使用 `VectorDelegates.TopRight Quadrant()` 来实现：

```
public static bool TopRightQuadrant(Vector target)
{
    if (target.Theta >= 0.0 && target.Theta <= 90.0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

如果方法的 `Vector` 参数值是介于  $0^\circ \sim 90^\circ$  之间的 `Theta` 值，该方法就返回 `true`，也就是说，它在前面的排序图中指向上或右。

在 `Main()` 方法中，通过 `Predicate<Vector>` 类型的委托使用这个方法，如下所示：

```
Predicate<Vector> searcher =
    new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));
Console.WriteLine(topRightQuadrantRoute.Sum());
```

这需要在 `Vectors` 中定义构造函数：

```
public Vectors(IEnumerable<Vector> initialItems)
{
    foreach (Vector vector in initialItems)
    {
        Add(vector);
    }
}
```

其中，使用 `IEnumerable<Vector>` 的接口初始化了一个新的 `Vectors` 集合，这是必需的，因为 `List<Vector>.FindAll()` 返回一个 `List<Vector>` 实例，而不是 `Vectors` 实例。

搜索的结果是，只返回 `Vector` 对象的一个子集，所以汇总的结果不同(这正是我们希望的)。使用这些泛型委托类型来排序和搜索泛型集合需要一段时间才能习惯，但代码更流畅、更高效了，代码的结构更富逻辑性。最好花点时间研究本节介绍的技术。

另外，在这个示例中，注意下面的代码：

```
Comparison<Vector> sorter = new Comparison<Vector>(
    VectorDelegates.Compare);
route.Sort(sorter);
```



可以简化为:

```
route.Sort(VectorDelegates.Compare);
```

这样就不需要隐式引用 `Comparison<Vector>` 类型了。实际上, 仍会创建这个类型的一个实例, 但它是隐式创建的。显然, `Sort()` 方法需要这个类型的实例才能工作, 但编译器会认识到这一点, 在我们提供的方法中自动创建该类型的实例。此时, 对 `VectorDelegates.Compare()` 的引用(没有括号)称为方法组。在许多情况下, 都可以使用方法组以这种方式隐式地创建委托, 使代码变得更容易读取。

### 3. Dictionary<K, V>

这个类型可以定义键/值对的集合。与本章前面介绍的其他泛型集合类型不同, 这个类需要实例化两个类型, 分别用于键和值, 以表示集合中的各个项。

实例化 `Dictionary<K, V>` 对象后, 就可以像在继承自 `DictionaryBase` 的类上那样, 对它执行相同的操作, 但要使用已有的类型安全的方法和属性。例如, 可以使用强类型化的 `Add()` 方法添加键/值对。

```
Dictionary<string, int> things = new Dictionary<string, int>();
things.Add("Green Things", 29);
things.Add("Blue Things", 94);
things.Add("Yellow Things", 34);
things.Add("Red Things", 52);
things.Add("Brown Things", 27);
```

可以使用 `Keys` 和 `Values` 属性迭代集合中的键和值:

```
foreach (string key in things.Keys)
{
    Console.WriteLine(key);
}

foreach (int value in things.Values)
{
    Console.WriteLine(value);
}
```

还可以迭代集合中的各个项, 把每个项作为一个 `KeyValuePair<K, V>` 实例来获取, 这与第 11 章介绍的 `DictionaryEntry` 对象十分相似:

```
foreach (KeyValuePair<string, int> thing in things)
{
    Console.WriteLine("{0} = {1}", thing.Key, thing.Value);
}
```

对于 `Dictionary<K, V>` 要注意的一点是, 每个项的键都必须是唯一的。如果要添加的项的键与已有项的键相同, 就会抛出 `ArgumentException` 异常。所以, `Dictionary<K, V>` 允许把 `IComparer<K>` 接口传递给其构造函数, 如果要把自己的类用作键, 且它们不支持 `IComparable` 或 `IComparable<K>` 接口, 或者要使用非默认的过程比较对象, 就必须把 `IComparer<K>` 接口传递给其构造函数。例如, 在上面的示例中, 可以使用不区分大小写的方法来比较字符串键:

```
Dictionary<string, int> things =
    new Dictionary<string, int>(StringComparer.CurrentCultureIgnoreCase);
```

如果使用下面的键，就会得到一个异常：

```
things.Add("Green Things", 29);
things.Add("Green things", 94);
```

也可以给构造函数传递初始容量(使用 `int`)或项的集合(使用 `IDictionary<K,V>` 接口)。

#### 4. 修改 CardLib，以使用泛型集合类

对前几章创建的 CardLib 项目可以进行简单的修改，即修改 Cards 集合类，以使用一个泛型集合类，这将减少许多代码行。对 Cards 的类定义需要做如下修改：



可从  
wrox.com  
下载源代码 }

```
public class Cards : List<Card>, ICloneable
{
    ...
}
```

代码段 Ch12CardLib\Cards.cs

还可以删除 Cards 的所有方法，但 `Clone()` 和 `CopyTo()` 除外，因为 `Clone()` 是 `ICloneable` 需要的方法，而 `List<Card>` 提供的 `CopyTo()` 版本处理的是 `Card` 对象数组，而不是 Cards 集合。需要对 `Clone()` 做一些轻微的修改，因为 `List<T>` 没有定义 `List` 属性：

```
public object Clone()
{
    Cards newCards = new Cards();
    foreach (Card sourceCard in this)
    {
        newCards.Add(sourceCard.Clone() as Card);
    }
    return newCards;
}
```

这里没有列出代码，因为这是很简单的修改，CardLib 的更新版本为 Ch12CardLib，它和第 11 章的客户代码包含在本章的下载代码中。

## 12.3 定义泛型类型

利用前面介绍的泛型知识，足以创建自己的泛型了。前面的许多代码都涉及到泛型类型，您还看到了多个使用泛型语法的实例。本节将定义如下内容：

- 泛型类
- 泛型接口
- 泛型方法
- 泛型委托

在定义泛型类型的过程中，还将讨论处理如下问题的一些更高级技术：

- `default` 关键字
- 约束类型

- 从泛型类中继承
- 泛型运算符

### 12.3.1 定义泛型类

要创建泛型类，只需在类定义中包含尖括号语法：

```
class MyGenericClass<T>
{
    ...
}
```

其中 T 可以是任意标识符，只要遵循通常的 C#命名规则即可，例如，不以数字开头等。但一般只使用 T。泛型类可以在其定义中包含任意多个类型，它们用逗号分隔开，例如：

```
class MyGenericClass<T1, T2, T3>
{
    ...
}
```

定义了这些类型之后，就可以在类定义中像使用其他类型那样使用它们。可以把它们用作成员变量的类型、属性或方法等成员的返回类型以及方法变元(argument)的参数类型(parameter type)等。例如：

```
class MyGenericClass<T1, T2, T3>
{
    private T1 innerT1Object;

    public MyGenericClass(T1 item)
    {
        innerT1Object = item;
    }

    public T1 InnerT1Object
    {
        get
        {
            return innerT1Object;
        }
    }
}
```

其中，类型 T1 的对象可以传递给构造函数，只能通过 InnerT1Object 属性对这个对象进行只读访问。注意，不能假定类提供了什么类型。例如，下面的代码就不会编译：

```
class MyGenericClass<T1, T2, T3>
{
    private T1 innerT1Object;

    public MyGenericClass()
    {
        innerT1Object = new T1();
    }
}
```

```

public T1 InnerT1Object
{
    get
    {
        return innerT1Object;
    }
}
}

```

我们不知道 T1 是什么，也就不能使用它的构造函数，它甚至可能没有构造函数，或者没有可公共访问的默认构造函数。如果不使用涉及本节后面介绍的高级技术的复杂代码，则只能对 T1 进行如下假设：可以把它看作继承自 `System.Object` 的类型或可以封箱到 `System.Object` 中的类型。

显然，这意味着不能对这个类型的实例进行非常有趣的操作，或者对为 `MyGenericClass` 泛型类提供的其他类型进行有趣的操作。不使用反射(这是用于在运行期间检查类型的高级技术，本章不介绍它)，就只能使用下面的代码：

```

public string GetAllTypesAsString()
{
    return "T1 = " + typeof(T1).ToString()
        + ", T2 = " + typeof(T2).ToString()
        + ", T3 = " + typeof(T3).ToString();
}

```

可以做一些其他工作，尤其是对集合进行操作，因为处理对象组是非常简单的，不需要对对象类型进行任何假设，这是为什么存在本章前面介绍的泛型集合类的一个原因。

另一个需要注意的限制是，在比较为泛型类型提供的类型值和 `null` 时，只能使用运算符 `==` 和 `!=`。例如，下面的代码会正常工作：

```

public bool Compare(T1 op1, T1 op2)
{
    if (op1 != null && op2 != null)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

其中，如果 T1 是一个值类型，则总是假定它是非空的，于是在上面的代码中，`Compare` 总是返回 `true`。但是，下面试图比较两个变元 `op1` 和 `op2` 的代码将不能编译：

```

public bool Compare(T1 op1, T1 op2)
{
    if (op1 == op2)
    {
        return true;
    }
    else
    {

```

```

        return false;
    }
}

```

其原因是这段代码假定 T1 支持 `==` 运算符。这说明，要对泛型进行实际的操作，需要更多地了解类中使用的类型。

## 1. default 关键字

要确定用于创建泛型类实例的类型，需要了解一个最基本的情况：它们是引用类型还是值类型。若不知道这个情况，就不能用下面的代码赋予 `null` 值：

```

public MyGenericClass()
{
    innerT1Object = null;
}

```

如果 T1 是值类型，则 `innerT1Object` 不能取 `null` 值，所以这段代码不会编译。幸好，开发人员考虑到了这个问题，使用 `default` 关键字(本书前面在 `switch` 结构中使用过它)的新用法解决了它。该新用法如下：

```

public MyGenericClass()
{
    innerT1Object = default(T1);
}

```

其结果是，如果 `innerT1Object` 是引用类型，就给它赋予 `null` 值；如果它是值类型，就给它赋予默认值。对于数字类型，这个默认值是 0；而结构根据其各个成员的类型，以相同的方式初始化为 0 或 `null`。`default` 关键字允许对必须使用的类型进行更多的操作，但为了更进一步，还需要限制所提供的类型。

## 2. 约束类型

前面用于泛型类的类型称为无绑定(`unbounded`)类型，因为没有对它们进行任何约束。而通过约束(`constraining`)类型，可以限制可用于实例化泛型类的类型，这有许多方式。例如，可以把类型限制为继承自某个类型。回顾前面使用的 `Animal`、`Cow` 和 `Chicken` 类，您可以把一个类型限制为 `Animal` 或继承自 `Animal`，则下面的代码是正确的：

```
MyGenericClass<Cow> = new MyGenericClass<Cow>();
```

但下面的代码不能编译：

```
MyGenericClass<string> = new MyGenericClass<string>();
```

在类定义中，这可以使用 `where` 关键字来实现：

```

class MyGenericClass<T> where T : constraint
{
    ...
}

```

其中 `constraint` 定义了约束。可以用这种方式提供许多约束，各个约束间用逗号分隔开：

```
class MyGenericClass<T> where T : constraint1, constraint2
{
    ...
}
```

还可以使用多个 `where` 语句，定义泛型类需要的任意类型或所有类型上的约束：

```
class MyGenericClass<T1, T2> where T1 : constraint1 where T2 : constraint2
{
    ...
}
```


约束必须出现在继承说明符的后面：

```
class MyGenericClass<T1, T2> : MyBaseClass, IMyInterface
    where T1 : constraint1 where T2 : constraint2
{
    ...
}
```

表 12-5 中列出了一些可用的约束。

表 12-5

约 束	定 义	用 法 示 例
<code>struct</code>	类型必须是值类型	在类中，需要值类型才能起作用，例如，T 类型的成员变量是 0，表示某种含义
<code>class</code>	类型必须是引用类型	在类中，需要引用类型才能起作用，例如，T 类型的成员变量是 <code>null</code> ，表示某种含义
<code>base-class</code>	类型必须是基类或继承自基类。可以给这个约束提供任意类名	在类中，需要继承自基类的某种基本功能，才能起作用
<code>interface</code>	类型必须是接口或实现了接口	在类中，需要接口公开的某种基本功能，才能起作用
<code>new()</code>	类型必须有一个公共的无参构造函数	在类中，需要能实例化 T 类型的变量，例如在构造函数中实例化



如果`new()`用作约束，它就必须是为类型指定的最后一个约束。

可以通过 `base-class` 约束，把一个类型参数用作另一个类型参数的约束，如下所示：

```
class MyGenericClass<T1, T2> where T2 : T1
{
    ...
}
```

其中，`T2` 必须与 `T1` 的类型相同，或者继承自 `T1`。这称为裸类型约束(`naked type constraint`)，表示一个泛型类型参数用作另一个类型参数的约束。


类型约束不能循环，例如：

```
class MyGenericClass<T1, T2> where T2 : T1 where T1 : T2
{
    ...
}
```

这段代码不能编译。下面的示例将定义和使用一个泛型类，该类使用前面几章介绍的 `Animal` 类系列。

试一试：定义泛型类

- (1) 在 `C:\BegVCSharp\Chapter12` 目录中创建一个新的控制台应用程序 `Ch12Ex04`。
- (2) 在 `Solution Explorer` 窗口中右击项目名称，选择 `Add | Add Existing Item` 选项。
- (3) 从 `C:\BegVCSharp\Chapter12\Ch12Ex02\Ch12Ex02` 目录中选择 `Animal.cs`、`Cow.cs` 和 `Chicken.cs` 文件，单击 `Add` 按钮。
- (4) 在已经添加的文件中修改名称空间声明，如下所示：




可从  
wrox.com  
下载源代码

```
namespace Ch12Ex04
```

---

Code snippets `Ch12Ex04\Animal.cs`、`Ch12Ex04\Cow.cs` 和 `Ch12Ex04\Chicken.cs`

- (5) 修改 `Animal.cs`，如下所示：



可从  
wrox.com  
下载源代码


```
public abstract class Animal
{
    ...

    public abstract void MakeANoise();
}
```

---

代码段 `Ch12Ex04\Animal.cs`

- (6) 修改 `Chicken.cs`，如下所示：



可从  
wrox.com  
下载源代码

```
public class Chicken : Animal
{
    ...

    public override void MakeANoise()
    {
        Console.WriteLine("{0} says 'cluck!'", name);
    }
}
```

---

代码段 `Ch12Ex04\Chicken.cs`

- (7) 修改 `Cow.cs`，如下所示：



可从  
wrox.com  
下载源代码

```
public class Cow : Animal
{
    ...
}
```



```

    public override void MakeANoise()
    {
        Console.WriteLine("{0} says 'moo!'", name);
    }
}

```

代码段 Ch12Ex04\Cow.cs

(8) 添加一个新类 SuperCow, 并修改 SuperCow.cs 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```

public class SuperCow : Cow
{
    public void Fly()
    {
        Console.WriteLine("{0} is flying!", name);
    }

    public SuperCow(string newName) : base(newName)
    {
    }

    public override void MakeANoise()
    {
        Console.WriteLine("{0} says 'here I come to save the day!'", name);
    }
}

```

代码段 Ch12Ex04\SuperCow.cs

(9) 添加一个新类 Farm, 并修改 Farm.cs 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch12Ex04
{
    public class Farm<T> : IEnumerable<T>
        where T : Animal
    {
        private List<T> animals = new List<T>();

        public List<T> Animals
        {
            get
            {
                return animals;
            }
        }

        public IEnumerator<T> GetEnumerator()
        {
            return animals.GetEnumerator();
        }
    }
}

```

```

    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return animals.GetEnumerator();
    }

    public void MakeNoises()
    {
        foreach (T animal in animals)
        {
            animal.MakeANoise();
        }
    }

    public void FeedTheAnimals()
    {
        foreach (T animal in animals)
        {
            animal.Feed();
        }
    }

    public Farm<Cow> GetCows()
    {
        Farm<Cow> cowFarm = new Farm<Cow>();
        foreach (T animal in animals)
        {
            if (animal is Cow)
            {
                cowFarm.Animals.Add(animal as Cow);
            }
        }
        return cowFarm;
    }
}
}

```

代码段 Ch12Ex04\Farm.cs

(10) 修改 Program.cs, 如下所示:



可从  
WTOX.COM  
下载源代码

```

static void Main(string[] args)
{
    Farm<Animal> farm = new Farm<Animal>();
    farm.Animals.Add(new Cow("Jack"));
    farm.Animals.Add(new Chicken("Vera"));
    farm.Animals.Add(new Chicken("Sally"));
    farm.Animals.Add(new SuperCow("Kevin"));
    farm.MakeNoises();

    Farm<Cow> dairyFarm = farm.GetCows();
    dairyFarm.FeedTheAnimals();

    foreach (Cow cow in dairyFarm)
    {

```



```

        if (cow is SuperCow)
        {
            (cow as SuperCow).Fly();
        }
    }
    Console.ReadKey();
}

```

代码段 Ch12Ex04\Program.cs

(11) 执行应用程序，结果如图 12-5 所示。

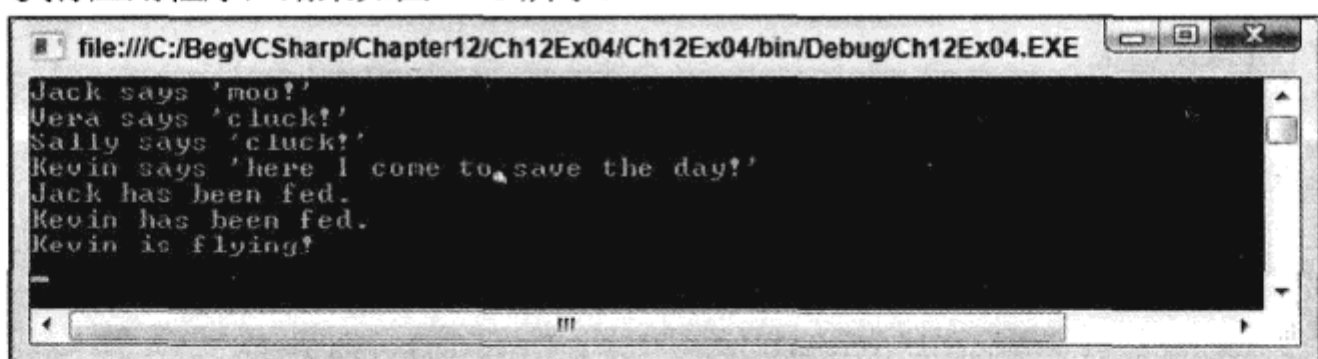


图 12-5

### 示例的说明

在这个示例中，创建了一个泛型类 `Farm<T>`，它没有继承泛型 `List` 类，而是将泛型 `list` 类作为公共属性公开，该 `List` 的类型由传送给 `Farm<T>` 的类型参数 `T` 确定，且被约束为 `Animal`，或者继承自 `Animal`。



可从  
wrox.com  
下载源代码

```

public class Farm<T> : IEnumerable<T>
    where T : Animal
{
    private List<T> animals = new List<T>();

    public List<T> Animals
    {
        get
        {
            return animals;
        }
    }
}

```

代码段 Ch12Ex04\Farm.cs

`Farm<T>` 还实现了 `IEnumerable<T>`，其中，`T` 传递给这个泛型接口，因此也以相同的方式进行了约束。实现这个接口，就可以迭代包含在 `Farm<T>` 中的项，而无需显式迭代 `Farm<T>.Animals`。很容易就能做到这一点，只需返回 `Animals` 公开的枚举器即可，该枚举器是一个 `List<T>` 类，也实现了 `IEnumerable<T>`。

```

public IEnumerator<T> GetEnumerator()
{
    return animals.GetEnumerator();
}

```

因为 `IEnumerable<T>` 继承自 `IEnumerable`，所以还需要实现 `IEnumerable.GetEnumerator()`：

```
IEnumerator IEnumerable.GetEnumerator()  
{  
    return animals.GetEnumerator();  
}
```

之后，`Farm<T>` 包含的两个方法利用了抽象类 `Animal` 的方法：

```
public void MakeNoises()  
{  
    foreach (T animal in animals)  
    {  
        animal.MakeANoise();  
    }  
}  
  
public void FeedTheAnimals()  
{  
    foreach (T animal in animals)  
    {  
        animal.Feed();  
    }  
}
```

`T` 被约束为 `Animal`，所以这段代码会正确编译——无论 `T` 是什么，都可以访问这些方法。

下一个方法 `GetCows()` 更加有趣。这个方法提取了集合中类型为 `Cow` (或继承自 `Cow`，例如，新的 `SuperCow` 类) 的所有项：

```
public Farm<Cow> GetCows()  
{  
    Farm<Cow> cowFarm = new Farm<Cow>();  
    foreach (T animal in animals)  
    {  
        if (animal is Cow)  
        {  
            cowFarm.Animals.Add(animal as Cow);  
        }  
    }  
    return cowFarm;  
}
```

有趣的是，这个方法似乎有点浪费。如果以后希望有同一系列的其他方法，如 `GetChickens()`，也需要显式实现它们。在使用许多类型的系统中，需要更多的方法。一个较好的解决方案是使用泛型方法，详见本章后面的内容。

`Program.cs` 中的客户代码测试了 `Farm` 的各个方法，它并没有包含前面列出的许多代码，所以不需要深入探讨这些代码。

### 3. 从泛型类中继承

上面示例中的 `Farm<T>` 类以及本章前面介绍的其他几个类都继承自一个泛型类型。在 `Farm<T>` 中，这个类型是一个接口 `IEnumerable<T>`。这里 `Farm<T>` 在 `T` 上提供的约束也会在 `IEnumerable<T>`

中使用的 `T` 上添加一个额外的约束。这可以用于限制未约束的类型，但是需要遵循一些规则。

首先，如果某个类型所继承的基类型中受到了约束，该类型就不能“解除约束”。也就是说，类型 `T` 在所继承的基类型中使用时，该类型必须受到至少与基类型相同的约束。例如，下面的代码是正确的：

```
class SuperFarm<T> : Farm<T>
    where T : SuperCow
{
}
```

因为 `T` 在 `Farm<T>` 中被约束为 `Animal`，把它约束为 `SuperCow`，就是把 `T` 约束为这些值的一个子集，所以这段代码可以正常运行。但是，不会编译以下代码：

```
class SuperFarm<T> : Farm<T>
    where T : struct
{
}
```

可以肯定地说，提供给 `SuperFarm<T>` 的类型 `T` 不能转换为可由 `Farm<T>` 使用的 `T`，所以代码不会编译。

甚至对于约束为超集的情况，也会出现相同的问题：

```
class SuperFarm<T> : Farm<T>
    where T : class
{
}
```

即使 `SuperFarm<T>` 允许有像 `Animal` 这样的类型，`Farm<T>` 中也不允许有满足类约束的其他类型。否则编译就会失败。这个规则适用于本章前面介绍的所有约束类型。

另外，如果继承了一个泛型类型，就必须提供所有必须的类型信息，这可以使用其他泛型类型参数的形式来提供，如上所述，也可以显式提供。这也适用于继承了泛型类型的非泛型类。例如：

```
public class Cards : List<Card>, ICloneable
{
}
```

这是可行的，但下面的代码会失败：

```
public class Cards : List<T>, ICloneable
{
}
```

因为没有提供 `T` 的信息，所以不能编译。



如果给泛型类型提供了参数，例如，上面的 `List<Card>`，就可以把类型引用为“关闭”。同样，继承 `List<T>`，就是继承一个“打开”的泛型类型。

#### 4. 泛型运算符

在 C# 中，可以像其他方法一样进行运算符的重写，这也可以在泛型类中实现此类重写。例如，可以在 `Farm<T>` 中定义如下隐式的转换运算符：

```
public static implicit operator List<Animal>(Farm<T> farm)
{
    List<Animal> result = new List<Animal>();
    foreach (T animal in farm)
    {
        result.Add(animal);
    }
    return result;
}
```

这样，如果需要，就可以在 `Farm<T>` 中把 `Animal` 对象直接作为 `List<Animal>` 来访问。例如，使用下面的运算符添加两个 `Farm<T>` 实例，这是很方便的：

```
public static Farm<T> operator +(Farm<T> farm1, List<T> farm2)
{
    Farm<T> result = new Farm<T>();

    foreach (T animal in farm1)
    {
        result.Animals.Add(animal);
    }
    foreach (T animal in farm2)
    {
        if (!result.Animals.Contains(animal))
        {
            result.Animals.Add(animal);
        }
    }
    return result;
}

public static Farm<T> operator +(List<T> farm1, Farm<T> farm2)
{
    return farm2 + farm1;
}
```

接着可以添加 `Farm<Animal>` 和 `Farm<Cow>` 的实例，如下所示：

```
Farm<Animal> newFarm = farm + dairyFarm;
```

在这行代码中，`dairyFarm` (是 `Farm<Cow>` 的实例) 隐式转换为 `List<Animal>`，`List<Animal>` 可以在 `Farm<T>` 中由重载运算符 `+` 使用。

读者可能认为，使用下面的代码也可以做到：

```
public static Farm<T> operator +(Farm<T> farm1, Farm<T> farm2)
{
    ...
}
```

但是, `Farm<Cow>` 不能转换为 `Farm<Animal>`, 所以汇总会失败。为了更进一步, 可以使用下面的转换运算符来解决这个问题:

```
public static implicit operator Farm<Animal>(Farm<T> farm)
{
    Farm<Animal> result = new Farm<Animal>();
    foreach (T animal in farm)
    {
        result.Animals.Add(animal);
    }
    return result;
}
```

使用这个运算符, `Farm<T>` 的实例(如 `Farm<Cow>`)就可以转换为 `Farm<Animal>` 的实例, 这解决了上面的问题。所以, 可以使用上面列出的两种方法, 但是后者更适合, 因为它比较简单。

### 5. 泛型结构

前几章说过, 结构实际上与类相同, 只有一些微小的区别, 而且结构是值类型, 不是引用类型。所以, 可以用与泛型类相同的方式来创建泛型结构。例如:

```
public struct MyStruct<T1, T2>
{
    public T1 item1;
    public T2 item2;
}
```

#### 12.3.2 定义泛型接口

前面介绍了几个泛型接口, 它们都位于 `Systems.Collections.Generic` 名称空间中, 例如, 上一个示例中使用的 `IEnumerable<T>`。定义泛型接口与定义泛型类所用的技术相同, 例如:

```
interface MyFarmingInterface<T>
    where T : Animal
{
    bool AttemptToBreed(T animal1, T animal2);

    T OldestInHerd { get; }
}
```

其中, 泛型参数 `T` 用作 `AttemptToBreed()` 的两个变元的类型和 `OldestInHerd` 属性的类型。

其继承规则与类相同。如果继承了一个基泛型接口, 就必须遵循“保持基接口泛型类型参数的约束”等规则。

#### 12.3.3 定义泛型方法

在上一个示例中提到了方法 `GetCows()`, 在讨论这个示例时也提到, 可以使用泛型方法得到这个方法的更一般形式。本节将说明如何达到这一目标。在泛型方法中, 返回类型和/或参数类型由泛型类型参数来确定。例如:

```
public T GetDefault<T>()
{
```



```
    return default(T);
}
```

这个小示例使用本章前面介绍的 `default` 关键字，为类型 `T` 返回默认值。这个方法的调用如下所示：

```
int myDefaultInt = GetDefault<T>();
```

在调用该方法时提供了类型参数 `T`。

这个 `T` 与用于给类提供泛型类型参数的类型差异极大。实际上，可以通过非泛型类来实现泛型方法：

```
public class Defaulter
{
    public T GetDefault<T>()
    {
        return default(T);
    }
}
```

但如果类是泛型的，就必须为泛型方法类型使用不同的标识符。下面的代码不会编译：

```
public class Defaulter<T>
{
    public T GetDefault<T>()
    {
        return default(T);
    }
}
```

必须重命名方法或类使用的类型 `T`。

泛型方法参数可以采用与类相同的方式使用约束，在此可以使用任意的类类型参数，例如：

```
public class Defaulter<T1>
{
    public T2 GetDefault<T2>()
        where T2 : T1
    {
        return default(T2);
    }
}
```

其中，为方法提供的类型 `T2` 必须与给类提供的 `T1` 相同，或者继承自 `T1`。这是约束泛型方法的常用方式。

在前面的 `Farm<T>` 类中，可以包含下面的方法(在 `Ch12Ex04` 的下载代码中包含它们，但已注释掉)。

```
public Farm<U> GetSpecies<U>() where U : T
{
    Farm<U> speciesFarm = new Farm<U>();
    foreach (T animal in animals)
    {
        if (animal is U)
```

```

    {
        speciesFarm.Animals.Add(animal as U);
    }
}
return speciesFarm;
}

```

这可以替代 `GetCows()` 和相同类型的其他方法。这里使用的泛型类型参数 `U` 由 `T` 约束，`T` 又由 `Farm<T>` 类约束为 `Animal`。因此，如果愿意，可以把 `T` 的实例视为 `Animal` 的实例。

在 `Ch12Ex04` 的客户代码 `Program.cs` 中，使用这个新方法需要进行一处修改：

```
Farm<Cow> dairyFarm = farm.GetSpecies<Cow>();
```

也可以编写如下代码：

```
Farm<Chicken> dairyFarm = farm.GetSpecies<Chicken>();
```

或者继承了 `Animal` 的其他类。

这里要注意，如果某个方法有泛型类型参数，会改变该方法的签名。也就是说，该方法有几个重载，它们仅在泛型类型参数上有区别。例如：

```

public void ProcessT<T>(T op1)
{
    ...
}

public void ProcessT<T, U>(T op1)
{
    ...
}

```

使用哪个方法取决于调用方法时指定的泛型类型参数的个数。

### 12.3.4 定义泛型委托

最后一个要介绍的泛型类型是泛型委托。本章前面在介绍如何排序和搜索泛型列表时曾介绍过它们，即分别为此使用了 `Comparison<T>` 和 `Predicate<T>` 委托。

第6章介绍了如何使用方法的参数和返回类型、`delegate` 关键字和委托名来定义委托，例如：

```
public delegate int MyDelegate(int op1, int op2);
```

要定义泛型委托，只需声明和使用一个或多个泛型类型参数，例如：

```
public delegate T1 MyDelegate<T1, T2>(T2 op1, T2 op2) where T1 : T2;
```

可以看出，也可以在这里使用约束。第13章将更详细地介绍委托，了解在常见的 C# 编程技术即“事件”中如何使用它们。

## 12.4 变体

变体(variance)是协变(covariance)和抗变(contravariance)的统称，这两个概念在 .NET 4 中引入。

实际上,它们已经存在不短的时间了(在.NET 2.0 中就可以使用),但直到.NET 4,仍然很难实现它们,因为它们需要定制的编译过程。

要掌握这些术语的含义,最简单的方式是把它们与多态性进行比较。多态性允许把派生类型的对象放在基类型的变量中,例如:

```
Cow myCow = new Cow("Geronimo");
Animal myAnimal = myCow;
```

其中把 Cow 类型的对象放在 Animal 类型的变量中,这是可行的,因为 Cow 派生自 Animal。但是,这不适用于接口,也就是说,下面的代码不能工作:

```
IMethaneProducer<Cow> cowMethaneProducer = myCow;
IMethaneProducer<Animal> animalMethaneProducer = cowMethaneProducer;
```

假定 Cow 支持 IMethaneProducer<Cow>接口,第一行代码就没有问题。但是,第二行代码预先假定两个接口类型有某种关系,但实际上这种关系不存在,所以无法把一种类型转换为另一种类型。肯定无法使用本章前面介绍的技术,因为泛型类型的所有类型参数都是不变的。但是可以在泛型接口和泛型委托上定义变体类型参数,以适合上述代码演示的情形。

为了使上述代码工作,IMethaneProducer<T>接口的类型参数 T 必须是协变的。有了协变的类型参数,就可以在 IMethaneProducer<Cow>和 IMethaneProducer<Animal>之间建立继承关系,这样一种类型的变量就可以包含另一种类型的值,这与多态性类似(但稍复杂些)。

为了完成对变体的介绍,需要看看变体的另一面:抗变。抗变和协变是类似的,但方向相反。抗变不能像协变那样,把泛型接口值放在使用基类型的变量中,而可以把该接口放在使用派生类型的变量中,例如:

```
IGrassMuncher<Cow> cowGrassMuncher = myCow;
IGrassMuncher<SuperCow> superCowGrassMuncher = cowGrassMuncher;
```

初看起来似乎有点古怪,因为不能通过多态性完成相同的功能。但是这在一些情况下是一项有效的技术,如“抗变”一节所述。

下面两节将介绍如何在泛型类型中实现变体,以及.NET Framework 如何使用变体简化编程。



本节所有代码都包含在演示项目 VarianceDemo 中,可供使用。

### 12.4.1 协变

要把泛型类型参数定义为协变,可以在类型定义中使用 out 关键字,如下面的示例所示:

```
public interface IMethaneProducer<out T>
{
    ...
}
```

对于接口定义,协变类型参数只能用作方法的返回值或属性 get 访问器。

说明协变用途的一个很好的例子在.NET Framework 中,即前面使用的 IEnumerable<T>接口。在

这个接口中，项类型 `T` 定义为协变，这表示可以把支持 `IEnumerable<Cow>` 的对象放在 `IEnumerable<Animal>` 类型的变量中。

因此下面的代码是有效的：

```
static void Main(string[] args)
{
    List<Cow> cows = new List<Cow>();
    cows.Add(new Cow("Geronimo"));
    cows.Add(new SuperCow("Tonto"));
    ListAnimals(cows);
    Console.ReadKey();
}

static void ListAnimals(IEnumerable<Animal> animals)
{
    foreach (Animal animal in animals)
    {
        Console.WriteLine(animal.ToString());
    }
}
```

其中 `cows` 变量的类型是 `List<Cow>`，它支持 `IEnumerable<Cow>` 接口。通过协变，这个变量可以传送给需要 `IEnumerable<Animal>` 类型的参数的方法。回想一下 `foreach` 循环的工作方式，就知道 `GetEnumerator()` 方法用于获取 `IEnumerator<T>` 的一个枚举器，该枚举器的 `Current` 属性用于访问项。`IEnumerator<T>` 还把其类型参数定义为协变，这表示可以把它用作参数的 `get` 访问器，而且一切都运转良好。

### 12.4.2 抗变

要把泛型类型参数定义为抗变，可以在类型定义中使用 `in` 关键字：

```
public interface IGrassMuncher<in T>
{
    ...
}
```

对于接口定义，抗变类型参数只能用作方法参数，不能用作返回类型。

理解这一点的最佳方式是列举一个在 .NET Framework 中使用抗变的例子。带有抗变类型参数的一个接口是前面用过的 `IComparer<T>`。可以给 `Animal` 实现这个接口，如下所示：

```
public class AnimalNameLengthComparer : IComparer<Animal>
{
    public int Compare(Animal x, Animal y)
    {
        return x.Name.Length.CompareTo(y.Name.Length);
    }
}
```

这个比较器按名称的长度比较动物，所以可以使用它对 `List<Animal>` 的实例排序。通过抗变，还可以使用它对 `List<Cow>` 的实例排序，尽管 `List<Cow>.Sort()` 方法需要 `IComparer<Cow>` 的实例。

```
List<Cow> cows = new List<Cow>();
```

```
cows.Add(new Cow("Geronimo"));
cows.Add(new SuperCow("Tonto"));
cows.Add(new Cow("Gerald"));
cows.Add(new Cow("Phil"));
cows.Sort(new AnimalNameLengthComparer());
```

大多数情况下, 抗变都会发生——它在 .NET Framework 中可帮助执行这种排序操作。 .NET 4 中这两种变体的优点是, 您可以在需要使用本节介绍的技术实现它。

## 12.5 小结

本章学习了如何在 C# 中使用泛型类型; 如何创建自己的泛型类型, 包括类、接口、方法和委托; 如何使用结构, 包括创建可空类型, 使用 `System.Collections.Generic` 名称空间中的类。

泛型是 C# 中一项功能极其强大的新技术, 使用它们创建的类可以同时达到多种目的, 并可以在许多不同的情况下使用。即使没有必要创建自己的泛型类型, 也可以使用泛型集合类。

第 13 章将研究其他基本知识, 探讨事件, 继续对基本 C# 语言的讨论。

## 12.6 练习

(1) 下面哪些元素可以是泛型?

- a. 类
- b. 方法
- c. 属性
- d. 运算符重载
- e. 结构
- f. 枚举

(2) 扩展 Ch12Ex01 中的 `Vector` 类, 使 `*` 运算符返回两个矢量的点积(dot product)。



两个矢量的点积定义为两个矢量的大小与两个矢量之间夹角余弦的乘积。

(3) 下面的代码存在什么错误? 请加以修改。

```
public class Instantiator<T>
{
    public T instance;

    public Instantiator()
    {
        instance = new T();
    }
}
```

(4) 下面的代码存在什么错误？请加以修改。

```
public class StringGetter<T>
{
    public string GetString<T>(T item)
    {
        return item.ToString();
    }
}
```

(5) 创建一个泛型类 `ShortCollection<T>`，它实现了 `ICollection<T>`，包含一个项集合及集合的最大容量。这个最大容量应是一个整数，并可以提供给 `ShortCollection<T>` 的构造函数，或者默认为 10。构造函数还应通过 `List<T>` 参数获取项的最初列表。该类与 `Collection<T>` 的功能相同，但如果试图给集合添加太多的项，或者传递给构造函数的 `List<T>` 包含太多的项，就会抛出 `IndexOutOfRangeException` 类型的异常。

(6) 下面的代码可以进行编译吗？试说明原因？

```
public interface IMethaneProducer<out T>
{
    void BelchAt(T target);
}
```

附录 A 给出了练习答案。

## 12.7 本章要点

主 题	重 要 概 念
使用泛型类型	泛型类型需要一个或多个类型参数才能工作。在声明变量时，传送需要的类型参数，就可以把泛型类型用作变量的类型。为此，应把逗号分隔的类型名列表放在尖括号中
可空类型	可空类型可以使用指定值类型的任意值或 <code>null</code> 值。使用 <code>Nullable&lt;T&gt;</code> 或 <code>T?</code> 语法，可以声明可空类型的变量
??运算符	空接合运算符返回第一个操作数的值，如果第一个操作数是 <code>null</code> ，就返回第二个操作数的值
泛型集合	泛型集合非常有用，因为它们内置了强类型化功能。可以使用 <code>List&lt;T&gt;</code> 、 <code>Collection&lt;T&gt;</code> 和 <code>Dictionary&lt;K, V&gt;</code> 等集合类型，它们还提供了泛型接口。为了针对泛型集合进行排序和搜索，应使用 <code>IComparer&lt;T&gt;</code> 和 <code>IComparable&lt;T&gt;</code> 接口
定义泛型类	泛型类型的定义十分类似于其他类型，但在指定类型名时需要添加泛型类型参数。与使用泛型类型一样，也需要把这些参数指定为逗号分隔的列表，并放在尖括号中。在使用类型名的地方都可以使用泛型类型参数，例如可以在方法的返回值和参数中使用它们
泛型类型的参数约束	为了高效地在泛型类型代码中使用泛型类型参数，可以在使用类型时约束可以提供的类型。可以根据基类、所支持的接口、是否必须是值类型或引用类型以及是否支持无参数的构造函数等，来约束类型参数。如果没有这些约束，就必须使用 <code>default</code> 关键字来实例化泛型类型的变量

(续表)

主 题	重 要 概 念
其他泛型类型	除类之外，还可以定义泛型接口、委托和方法
变体	变体是类似于多态性的一个概念，但应用于类型参数。它允许使用一个泛型类型替代另一个泛型类型，这些泛型类型仅在所使用的泛型类型参数上有区别。协变允许在两种类型之间转换，其中目标类型有一个类型参数，它是源类型的类型参数的基类。抗变允许进行相反的转换。协变类型参数用 out 参数定义，只能用作返回类型和属性 get 访问器的类型。抗变类型参数用 in 参数定义，只能用作方法的参数





# 第 13 章

## 其他 OOP 技术

### 本章内容:

---

- ::运算符
- 全局名称空间限定符
- 如何创建定制异常
- 如何使用事件
- 如何使用匿名方法

本章将介绍前面未涉及的内容，继续对 C# 语言的讨论。并不是说这些技术没用，它们只是不适合放在前面的主题中讨论而已。

本章还将对前面几章构建的 CardLib 代码进行最后的修改，并使用 CardLib 来创建扑克牌游戏。

### 13.1 ::运算符和全局名称空间限定符

::运算符提供了另一种访问名称空间中类型的方式。如果要使用一个名称空间的别名，但该别名与实际名称空间层次结构之间的界限不清晰，这将是必要的。在那种情况下，名称空间层次结构优先于名称空间别名。为了阐明其含义，考虑下列代码：

```
using MyNamespaceAlias = MyRootNamespace.MyNestedNamespace;

namespace MyRootNamespace
{
    namespace MyNamespaceAlias
    {
        public class MyClass
        {
        }
    }
}

namespace MyNestedNamespace
```

```

    {
        public class MyClass
        {
        }
    }
}

```

**MyRootNamespace** 中的代码使用下面的代码引用一个类:

```
MyNamespaceAlias.MyClass
```

这行代码表示的类是 **MyRootNamespace.MyNamespaceAlias.MyClass** 类, 而不是 **MyRootNamespace.MyNestedNamespace.MyClass** 类。也就是说, **MyRootNamespace.MyNamespaceAlias** 名称空间隐藏了由 **using** 语句定义的别名, 该别名指向 **MyRootNamespace.MyNestedNamespace** 名称空间。仍然可以访问这个名称空间以及其中包含的类, 但需要使用不同的语法:

```
MyNestedNamespace.MyClass
```

另外, 还可以使用 **::** 运算符:

```
MyNamespaceAlias::MyClass
```

使用这个运算符会迫使编译器使用由 **using** 语句定义的别名, 因此代码指向 **MyRootNamespace.MyNestedNamespace.MyClass**。

**::** 运算符还可以和 **global** 关键字一起使用, 它实际上是顶级根名称空间的别名。这有助于更清晰地说明要指向哪个名称空间, 如下所示:

```
global::System.Collections.Generic.List<int>
```

这是希望使用的类, 即 **List<T>** 泛型集合类。它肯定不是用下列代码定义的类:

```

namespace MyRootNamespace
{
    namespace System
    {
        namespace Collections
        {
            namespace Generic
            {
                class List<T>
                {
                }
            }
        }
    }
}

```

当然, 应避免使名称空间的名称与已有的 .NET 名称空间相同, 但这个问题只在大型项目中才会出现, 尤其是作为大型开发队伍中的一员进行开发时, 此类问题就更严重。使用 **::** 运算符和 **global** 关键字可能是访问所需类型的唯一方式。

## 13.2 定制异常

第 7 章讨论了异常,以及如何使用 `try...catch...finally` 块处理它们。我们还论述了几个标准的 .NET 异常,包括异常的基类 `System.Exception`。在应用程序中,有时也可以从这个基类中派生自己的异常类,并使用它们,而不是使用标准的异常。这样就可以把更具体的信息发送给捕获该异常的代码,让处理异常的捕获代码更有针对性。例如,可以给异常类添加一个新属性,以便访问某些底层信息,这样异常的接收代码就可以做出必要的改变,或者仅给出异常起因的更多信息。

定义了异常类后,就可以使用 `Debug | Exceptions` 对话框中的 `Add` 按钮,把它添加到 VS 可以识别的异常列表中,然后定义与异常相关的操作,如第 7 章所述。



在 `System` 名称空间中有两个基本的异常类 `ApplicationException` 和 `SystemException`, 它们派生于 `Exception`。 `SystemException` 用作 .NET Framework 预定义的异常的基类, `ApplicationException` 由开发人员用于派生自己的异常类。但最近的最佳实践方式是不从这个类中派生异常,而应使用 `Exception`。 `ApplicationException` 类在未来可能会被废弃。

### 给 CardLib 添加定制异常

定制异常的用法最好通过升级 `CardLib` 项目来说明。如果试图访问索引小于 0 或大于 51 的扑克牌, `Deck.GetCard()` 方法目前就会抛出一个标准的 .NET 异常,但下面改为使用一个定制异常。

首先,需要在 `BegVCSharp\Chapter13` 目录中创建一个新的类库项目 `Ch13CardLib`, 像以前一样把类从 `Ch12CardLib` 中复制过来,并把名称空间改为 `Ch13CardLib`。接着定义该异常。方法是使用在新类文件 `CardOutOfRangeException.cs` 中定义的一个新类,这个新类是使用 `Project | Add Class` 添加到 `Ch13CardLib` 项目中的:



可从  
wrox.com  
下载源代码

```
public class CardOutOfRangeException : Exception
{
    private Cards deckContents;

    public Cards DeckContents
    {
        get
        {
            return deckContents;
        }
    }

    public CardOutOfRangeException(Cards sourceDeckContents) :
        base("There are only 52 cards in the deck.")
    {
        deckContents = sourceDeckContents;
    }
}
```

代码段 `Ch13CardLib\CardOutOfRangeException.cs`

这个类的构造函数需要使用 Cards 类的一个实例，它允许通过 DeckContents 属性来访问这个 Cards 对象，为 Exception 基构造函数提供合适的错误信息，使该错误信息可以通过类的 Message 属性得到。

接着，在 Deck.cs 中添加抛出该异常的代码(替换原来的标准异常):



可从  
wrox.com  
下载源代码

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
        return cards[cardNum];
    else
        throw new CardOutOfRangeException(cards.Clone() as Cards);
}
```

代码段 Ch13CardLib\Deck.cs

DeckContents 属性是通过对 Deck 对象的当前内容(其形式是一个 Cards 对象)进行深度复制来初始化的。这表示，此时的内容是异常抛出时的内容，所以以后对 Deck 内容的修改不会丢失这些信息。

要进行测试，使用下面的客户代码(在本章的下载代码的 Ch13CardClient 中):



可从  
wrox.com  
下载源代码

```
Deck deck1 = new Deck();
try
{
    Card myCard = deck1.GetCard(60);
}
catch (CardOutOfRangeException e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine(e.DeckContents[0]);
}
Console.ReadKey();
```

代码段 Ch13CardClient\Program.cs

结果如图 13-1 所示。

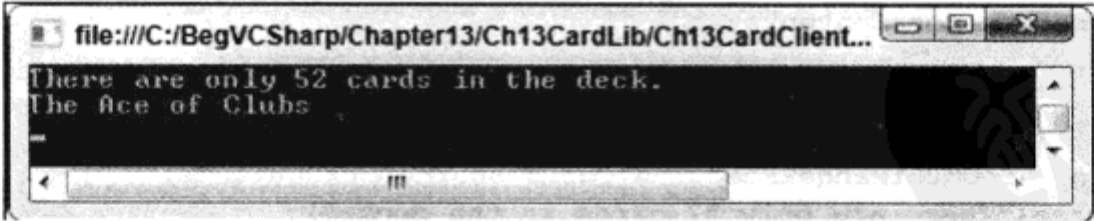


图 13-1

其中捕获代码把异常的 Message 属性写到屏幕上。我们还通过 DeckContents 显示了 Cards 对象中的第一张牌，以证明可以通过定制的异常对象访问 Cards 集合。

## 13.3 事件

本节主要讨论.NET 中最常用的 OOP 技术：事件。像往常一样，先介绍基础知识，分析事件到底是什么。之后讨论几个简单的事件，看看使用它们可以做什么。然后论述如何创建和使用自己的事件。

本章的最后介绍如何给 CardLib 类库添加一个事件，使该类库更完整。另外，因为这是在介绍一些更高级论题之前的最后一部分，我们还将创建一个使用该类库的有趣的扑克牌游戏应用程序。

### 13.3.1 事件的含义

事件类似于异常，因为它们都由对象引发(抛出)，我们可以提供代码来处理事件。但它们也有几个重要的区别。最重要的区别是并没有与 try...catch 类似的结构来处理事件，而必须订阅(subscribe)它们。订阅一个事件的含义是提供代码，在事件发生时执行这些代码，它们称为事件处理程序。

单个事件可供多个处理程序订阅，在该事件发生时，这些处理程序都会被调用，其中包括引发该事件的对象所在的类中的事件处理程序，但事件处理程序也可能在其他类中。

事件处理程序本身都是简单的方法。对事件处理方法的唯一限制是它必须匹配于事件所要求的返回类型和参数。这个限制是事件定义的一部分，由一个委托指定。



在事件中使用委托是非常有用的。第 6 章对此已进行了论述，读者可以温习这一部分，复习一下委托是什么以及如何使用它们。

基本处理过程如下所示：首先，应用程序创建一个可以引发事件的对象。例如，假定一个即时消息传送(instant messaging)应用程序创建的对象表示一个远程用户的连接。当接收到通过该连接从远程用户传送来的信息时，这个连接对象会引发一个事件，如图 13-2 所示。



图 13-2

接着，应用程序订阅事件。为此，即时消息传送应用程序将定义一个方法，该方法可以与事件指定的委托类型一起使用，把这个方法的一个引用传送给事件，而事件的处理方法可以是另一个对象的方法，假定是表示显示设备的对象，当接收到信息时，该方法将显示即时消息，如图 13-3 所示。

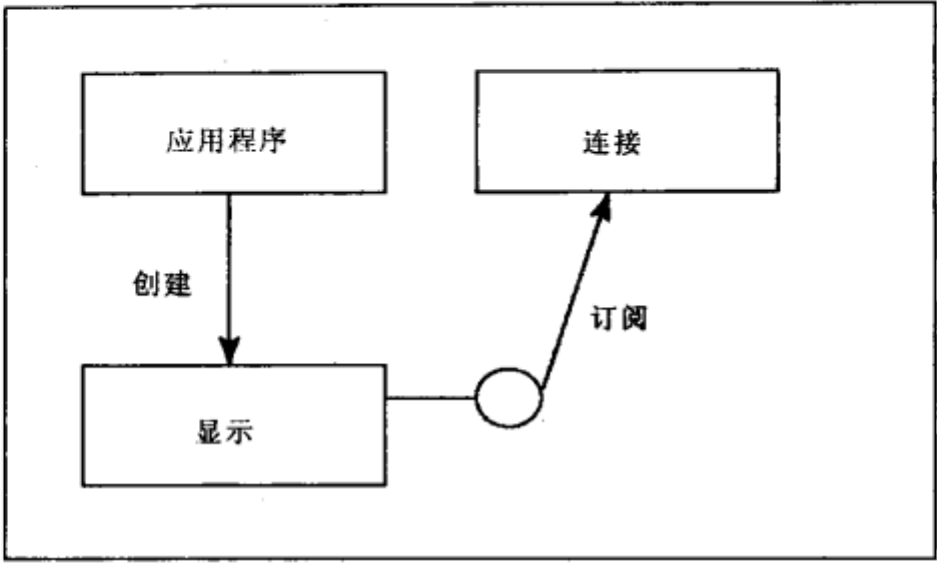


图 13-3

引发事件后，就通知订阅器。当接收到通过连接对象传来的即时消息时，就调用显示设备对象上的事件处理方法。因为我们使用的是一个标准方法，所以引发事件的对象可以通过参数传送任何相关的信息，这样就大大增加了事件的通用性。在本例中，一个参数是即时消息的文本，事件处理程序可以在显示设备对象上显示它，如图 13-4 所示。

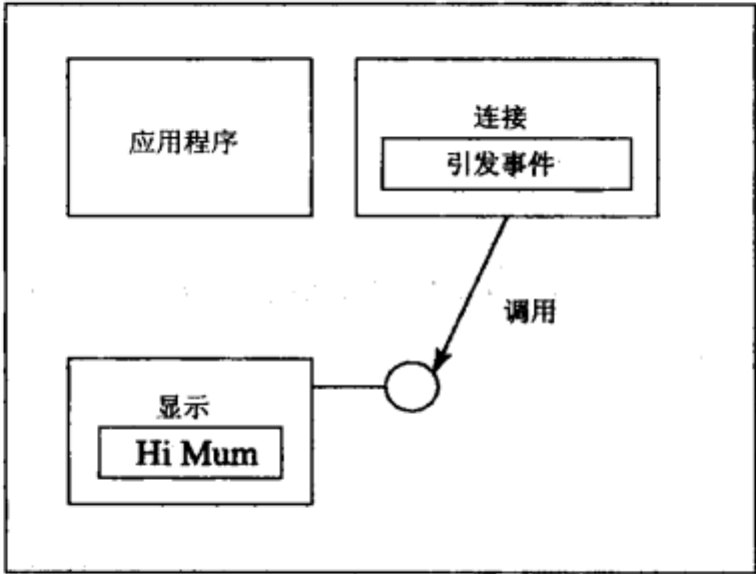


图 13-4

13.3.2 处理事件

如前所述，要处理事件，需要提供一个事件处理方法来订阅事件，该方法的返回类型和参数应该匹配事件指定的委托。下面的示例使用一个简单的计时器对象引发事件，调用一个处理方法。

试一试：处理事件

- (1) 在 C:\BegVCSharp\Chapter13 目录中下创建一个新的控制台应用程序 Ch13Ex01。
- (2) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

using System.Timers;

namespace Ch13Ex01
{
    class Program
    {
        static int counter = 0;

        static string displayString =
            "This string will appear one letter at a time. ";
        static void Main(string[] args)
        {
            Timer myTimer = new Timer(100);
            myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
            myTimer.Start();
            Console.ReadKey();
        }

        static void WriteChar(object source, ElapsedEventArgs e)
        {
            Console.Write(displayString[counter++ % displayString.Length]);
        }
    }
}

```

代码段 Ch13Ex01\Program.cs

(3) 运行应用程序(启动后, 按回车键将终止程序的执行), 在经过短暂运行后, 将显示如图 13-5 所示的结果。

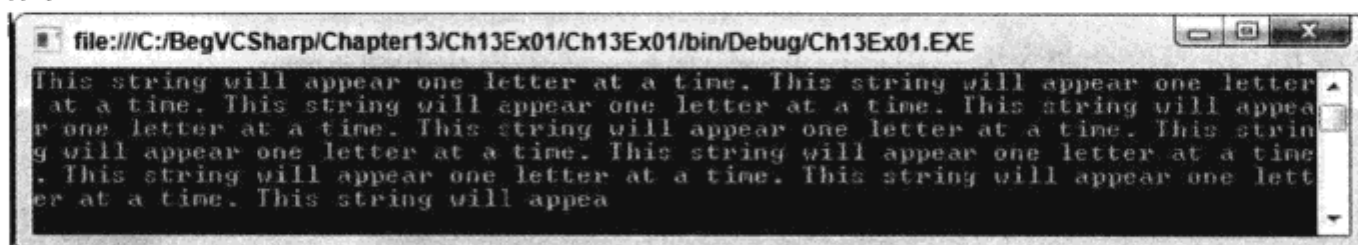


图 13-5

### 示例的说明

用于引发事件的对象是 `System.Timers.Timer` 类的一个实例。使用一个时间段(以毫秒为单位)来初始化该对象。当使用 `Start()` 方法启动 `Timer` 对象时, 就引发一系列事件, 根据指定的时间段来引发事件。`Main()` 用 100 毫秒初始化 `Timer` 对象, 所以在启动该对象后, 1 秒钟内将引发 10 次事件:

```

static void Main(string[] args)
{
    Timer myTimer = new Timer(100);

```

`Timer` 对象有一个 `Elapsed` 事件, 这个事件要求事件处理程序必须匹配 `System.Timers.ElapsedEventHandler` 委托类型的返回类型和参数, 该委托是 .NET Framework 中定义的标准委托之一, 指定了返回类型和参数:

```

void functionName(object source, ElapsedEventArgs e);

```



`Timer` 对象的第一个参数是它本身的引用，第二个参数则是 `ElapsedEventArgs` 对象的一个实例。现在可以不考虑这些参数，后面将论述它们。

在代码中，有一个匹配该返回类型和参数的方法：

```
static void WriteChar(object source, ElapsedEventArgs e)
{
    Console.Write(displayString[counter++ % displayString.Length]);
}
```

这个方法使用 `Class1` 的两个静态字段 `counter` 和 `displayString` 来显示一个字符。每次调用方法时，显示的字符都不相同。

下一个任务是把这个处理程序与事件关联起来——即订阅它。为此，可以使用 `+=` 运算符，给事件添加一个处理程序，其形式是使用事件处理方法初始化的一个新委托实例：

```
static void Main(string[] args)
{
    Timer myTimer = new Timer(100);
    myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
}
```

这个命令(使用有点古怪的语法，专用于委托)在列表中添加一个处理程序，当引发 `Elapsed` 事件时，就会调用该处理程序。可以给这个列表添加任意多个处理程序，只要它们满足指定的条件即可。当引发事件时，会依次调用每个处理程序。

`Main()` 剩下的任务是启动计时器：

```
myTimer.Start();
```

我们不想在处理完任何事件前终止应用程序，所以要让 `Main()` 函数一直执行。最简单的方式是请求用户输入，因为这个命令要在用户按下回车键后，才会停止处理。

```
Console.ReadKey();
```

在这里，`Main()` 中的处理会停止，但 `Timer` 对象中的处理将继续。当该对象引发事件时，就调用 `WriteChar()` 方法，同时该方法运行 `Console.ReadLine()` 语句。

注意，可以使用上一章介绍的方法组概念简化添加事件处理程序的语法：

```
myTimer.Elapsed += WriteChar;
```

最终结果是相同的，但不必显式指定委托类型，编译器会根据使用事件的上下文来指定它。但是，许多程序员不喜欢这个语法，因为它降低了可读性——不再能一眼看出使用了什么委托类型。如果喜欢，就可以使用这个语法。但为了清晰起见，本章使用的所有委托都显式指定。

### 13.3.3 定义事件

接着论述如何定义和使用自己的事件。我们将使用本节前言介绍的即时消息传送应用程序示例，并创建一个 `Connection` 对象，该对象引发由 `Display` 对象处理的事件。

#### 试一试：定义事件

- (1) 在 `C:\BegVCSharp\Chapter13` 目录中创建一个新控制台应用程序 `Ch13Ex02`。

(2) 添加一个新类 Connection, 并修改 Connection.cs, 如下所示:



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Timers;

namespace Ch13Ex02
{
    public delegate void MessageHandler(string messageText);

    public class Connection
    {
        public event MessageHandler MessageArrived;
        private Timer pollTimer;

        public Connection()
        {
            pollTimer = new Timer(100);
            pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
        }

        public void Connect()
        {
            pollTimer.Start();
        }

        public void Disconnect()
        {
            pollTimer.Stop();
        }

        private static Random random = new Random();

        private void CheckForMessage(object source, ElapsedEventArgs e)
        {
            Console.WriteLine("Checking for new messages.");
            if ((random.Next(9) == 0) && (MessageArrived != null))
            {
                MessageArrived("Hello Mum!");
            }
        }
    }
}
```

代码段 Ch13Ex02\Connection.cs

(3) 添加一个新类 Display, 并修改 Display.cs, 如下:



可从  
wrox.com  
下载源代码

```
namespace Ch13Ex02
{
    public class Display
    {
        public void DisplayMessage(string message)
```

```
    {  
        Console.WriteLine("Message arrived: {0}", message);  
    }  
}  
}
```

代码段 Ch13Ex02\Display.cs

(4) 修改 Program.cs 中的代码，如下所示：



```
static void Main(string[] args)  
{  
    Connection myConnection = new Connection();  
    Display myDisplay = new Display();  
    myConnection.MessageArrived +=  
        new MessageHandler (myDisplay.DisplayMessage);  
    myConnection.Connect();  
    Console.ReadKey();  
}
```

代码段 Ch13Ex02\Program.cs

(5) 运行应用程序，其结果如图 13-6 所示。

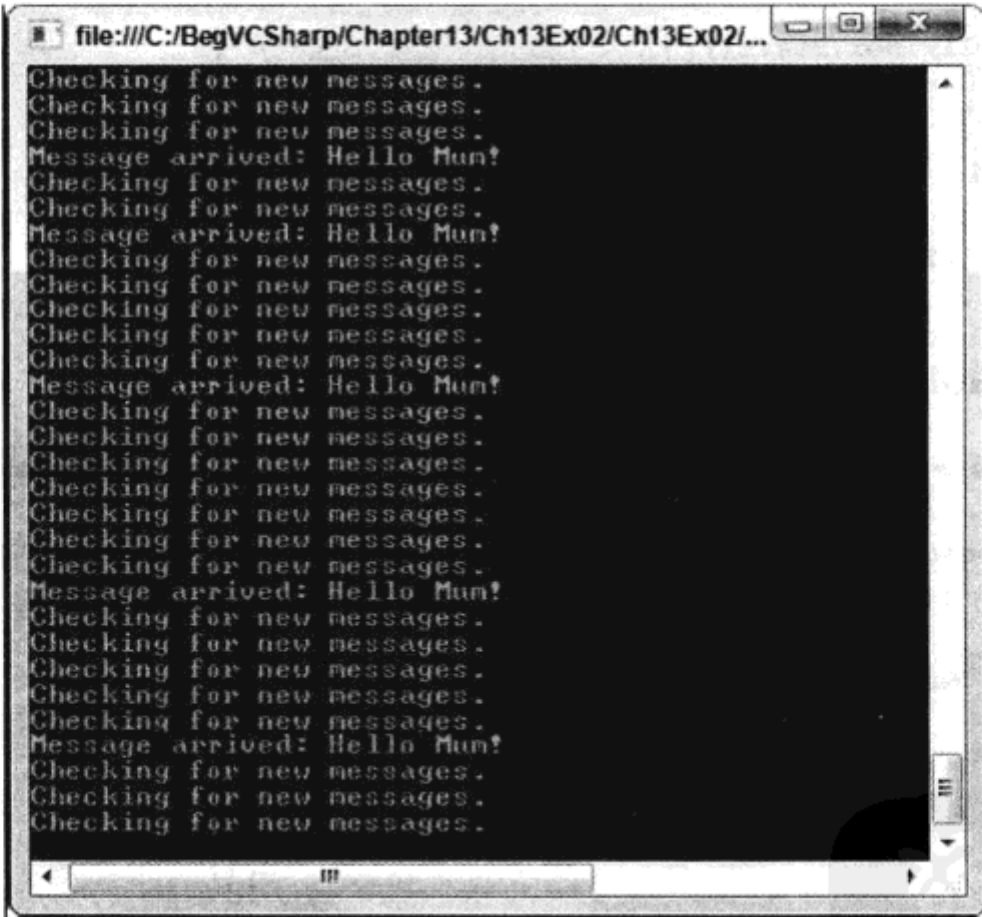


图 13-6

示例的说明

在这个应用程序中，大部分工作是由 Connection 类完成的。这个类的实例使用如本章第一个示例中所示的 Timer 对象，在类的构造函数中初始化它，并通过 Connect()和 Disconnect()访问它的状态(可访问和禁止访问)：

```

public class Connection
{
    private Timer pollTimer;

    public Connection()
    {
        pollTimer = new Timer(100);
        pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
    }

    public void Connect()
    {
        pollTimer.Start();
    }

    public void Disconnect()
    {
        pollTimer.Stop();
    }

    ...
}

```

在构造函数中，我们还以与第一个示例相同的方式注册了 `Elapsed` 事件的一个事件处理程序。每当调用这个处理程序方法 `CheckForMessage()` 的次数达到 10 次后，就会引发一个事件。在分析它的代码前，先看看事件的定义。

在定义事件前，必须先定义一个委托类型，以用于该事件，这个委托类型指定了事件处理方法必须拥有的返回类型和参数。为此，我们使用标准的委托语法，在 `Ch13Ex02` 名称空间中将该委托定义为公共类型，使该类型可供外部代码使用：

```

namespace Ch13Ex02
{
    public delegate void MessageHandler(string messageText);
}

```

这个委托类型称为 `MessageHandler`，是 `void` 函数的签名，它有一个 `string` 参数。使用这个参数可以把 `Connection` 对象收到的即时消息发送给 `Display` 对象。定义了委托 (或者找到合适的现有委托) 后，就可以把事件本身定义为 `Connection` 类的一个成员：

```

public class Connection
{
    public event MessageHandler MessageArrived;
}

```

给事件命名(这里使用名称 `MessageArrived`)，在声明时，使用 `event` 关键字，并指定要使用的委托类型(前面定义的 `MessageHandler` 委托类型)。以这种方式声明了事件后，就可以引发它，方法是按名称来调用它，就好像它是一个其返回类型和参数是由委托指定的方法一样。例如，使用下面的代码引发这个事件：

```

MessageArrived("This is a message.");

```

如果定义该委托时不包含任何参数，就可以使用下面的代码：

```
MessageArrived();
```

如果定义了较多参数，就需要用比较多的代码来引发事件。CheckForMessage()方法如下所示：

```
private static Random random = new Random();

private void CheckForMessage(object source, ElapsedEventArgs e)
{
    Console.WriteLine("Checking for new messages.");
    if ((random.Next(9) == 0) && (MessageArrived != null))
    {
        MessageArrived("Hello Mum!");
    }
}
```

使用前面几章中的 Random 类实例，生成一个 0~9 之间的随机数，如果该随机数为 0，就引发一个事件，它的发生几率为 10%。这类似于轮流检测连接，看看是否接收到消息，不可能每次检测时，都没有接收到消息。为了把计时器与 Connection 的实例分隔开，使用了 Random 类的一个私有静态实例。

注意，这里还提供了其他逻辑。只有表达式 MessageArrived != null 为 true，才引发一个事件。这个表达式也使用了委托语法，但语法略有不同，其含义是“事件是否有订阅者？”。如果没有订阅者，MessageArrived 就是 null，也就不会引发事件。

订阅事件的类是 Display，它包含一个方法 DisplayMessage()，其定义如下所示：

```
public class Display
{
    public void DisplayMessage(string message)
    {
        Console.WriteLine("Message arrived: {0}", message);
    }
}
```

这个方法匹配于委托类型(而且是公共的，如果类不是生成该事件的类，则其事件处理程序就必须是公共的)，所以可以用它来响应 MessageArrived 事件。

剩下的是 Main()中的代码初始化了 Connection 和 Display 类的实例，把它们关联起来，开始执行任务。这里需要的代码类似于第一个示例中的代码：

```
static void Main(string[] args)
{
    Connection myConnection = new Connection();
    Display myDisplay = new Display();
    myConnection.MessageArrived +=
        new MessageHandler(myDisplay.DisplayMessage);
    myConnection.Connect();
    Console.ReadKey();
}
```

再次调用 Console.ReadKey()，当开始执行 Connection 对象的 Connect()方法时，暂停 Main()的处理。

## 1. 多用途的事件处理程序

前面 `Timer.Elapsed` 事件的委托包含了事件处理程序中常见的两类参数，如下所示：

- `object source`——引发事件的对象的引用
- `ElapsedEventArgs`——由事件传送的参数

在这个事件(以及许多其他的事件)中使用 `Object` 类型参数的原因是，我们常常要为由不同对象引发的几个相同事件使用同一个事件处理程序，但仍要指定哪个对象生成了事件。

要说明这一点，下面将扩展上一个示例。

**试一试：使用多用途的事件处理程序**

(1) 在 `C:\BegVCSharp\Chapter13` 目录中创建一个新控制台应用程序 `Ch13Ex03`。

(2) 复制 `Ch13Ex02` 中 `Program.cs`、`Connection.cs` 和 `Display.cs` 的代码，并把每个文件中的 `Ch13Ex02` 名称空间改成 `Ch13Ex03`。

(3) 添加一个新类 `MessageArrivedEventArgs`，修改 `MessageArrivedEventArgs.cs`，如下所示：



可从  
wrox.com  
下载源代码

```
namespace Ch13Ex03
{
    public class MessageArrivedEventArgs : EventArgs
    {
        private string message;

        public string Message
        {
            get
            {
                return message;
            }
        }

        public MessageArrivedEventArgs()
        {
            message = "No message sent.";
        }

        public MessageArrivedEventArgs(string newMessage)
        {
            message = newMessage;
        }
    }
}
```

代码段 Ch13Ex03\MessageArrivedEventArgs.cs

(4) 修改 `Connection.cs`，如下所示：



可从  
wrox.com  
下载源代码

```
namespace Ch13Ex03
{
    public delegate void MessageHandler(Connection source,
                                        MessageArrivedEventArgs e);
}
```

```

public class Connection
{
    public event MessageHandler MessageArrived;

    private string name { get; set; }
    ...

    private void CheckForMessage(object source, EventArgs e)
    {
        Console.WriteLine("Checking for new messages.");
        if ((random.Next(9) == 0) && (MessageArrived != null))
        {
            MessageArrived(this, new MessageArrivedEventArgs("Hello Mum!"));
        }
    }
    ...
}

```

---

代码段 Ch13Ex03\Connection.cs

---

(5) 修改 Display.cs(包括事件参数类型), 如下所示:



```

public void DisplayMessage(Connection source, MessageArrivedEventArgs e)
{
    Console.WriteLine("Message arrived from: {0}", source.Name);
    Console.WriteLine("Message Text: {0}", e.Message);
}

```

---

代码段 Ch13Ex03\Display.cs

---

(6) 修改 Program.cs, 如下所示:



```

static void Main(string[] args)
{
    Connection myConnection1 = new Connection();
    myConnection1.Name = "First connection.";
    Connection myConnection2 = new Connection();
    myConnection2.Name = "Second connection.";
    Display myDisplay = new Display();
    myConnection1.MessageArrived +=
        new MessageHandler(myDisplay.DisplayMessage);
    myConnection2.MessageArrived +=
        new MessageHandler(myDisplay.DisplayMessage);
    myConnection1.Connect();
    myConnection2.Connect();
    Console.ReadKey();
}

```

---

代码段 Ch13Ex03\Program.cs

---



(7) 运行应用程序，其结果如图 13-7 所示。

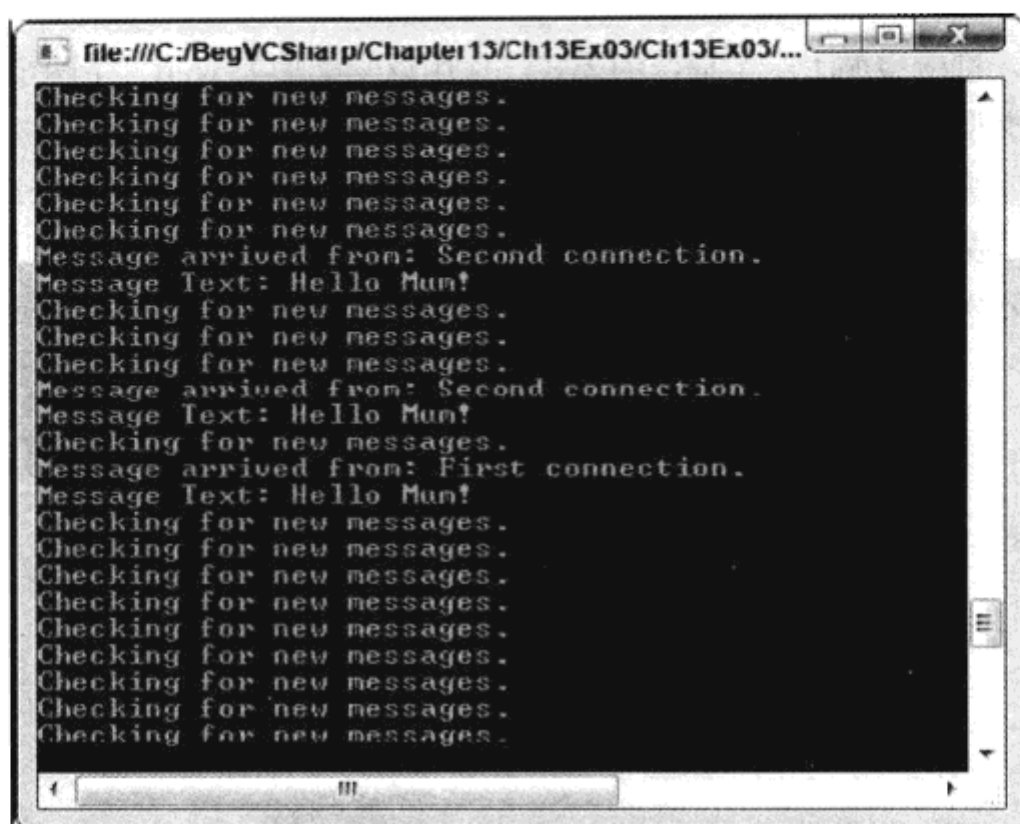


图 13-7

#### 示例的说明

发送一个引发事件的对象引用，将其作为事件处理程序的一个参数，就可以为不同的对象定制处理程序的响应。利用该引用可以访问源对象，包括它的属性。

通过发送包含在派生于 `System.EventArgs` (与 `ElapsedEventArgs` 相同) 的类中的参数，就可以将其其他必要信息提供为参数(例如，`MessageArrivedEventArgs` 类上的 `Message` 参数)。

另外，这些参数也将得益于多态性。可以为 `MessageArrived` 事件定义一个处理程序，如下所示：

```
public void DisplayMessage(object source, EventArgs e)
{
    Console.WriteLine("Message arrived from: {0}",
        ((Connection)source).Name);
    Console.WriteLine("Message Text: {0}",
        ((MessageArrivedEventArgs)e).Message);
}
```

修改 `Connection.cs` 中的委托定义，如下所示：

```
public delegate void MessageHandler(object source, EventArgs e);
```

这个应用程序将像以前那样执行，但 `DisplayMessage()` 方法变得更加通用(至少从理论上讲是这样的——需要使用更多实现代码，才能满足生产环境的质量要求)。这个处理程序还可以处理其他事件，例如 `Timer.Elapsed` 事件，但必须修改处理程序的内部代码，这样，在引发这个事件时，发送过来的参数才会得到正确处理(以这种方式把它们转换为 `Connection` 和 `MessageArrivedEventArgs` 对象，会抛出一个异常，所以这里应使用 `as` 运算符，检查 `null` 值)。

## 2. EventHandler 和泛型 EventHandler<T>类型

大多数情况下，都应遵循上一节提出的模式，使用返回类型为 void、带两个参数的事件处理程序。第一个参数的类型是 object，是事件源。第二个参数的类型派生于 System.EventArgs，包含任意事件变元。这非常常见，所以.NET 提供了两个委托类型 EventHandler 和 EventHandler<T>，以便于定义事件。它们都是委托，使用标准的事件处理模式。泛型版本允许指定要使用的事件变元的类型。

所以在前面的示例中，可以不定义自己的 MessageHandler 泛型类型，而是定义 MessageArrived 事件，如下所示：

```
public class Connection
{
    public event EventHandler MessageArrived;

    ...
}
```

甚或：

```
public class Connection
{
    public event EventHandler<MessageArrivedEventArgs> MessageArrived;

    ...
}
```

这显然是件好事，因为它简化了代码。

## 3. 返回值和事件处理程序

前面的所有事件处理程序都使用 void 类型的返回值。可以为事件提供返回类型，但这会出问题。这是因为引发给定的事件，可能会调用好几个事件处理程序。如果这些处理程序都返回一个值，那么我们该使用哪个返回值？

系统处理这个问题的方式是，只允许访问由事件处理程序最后返回的那个值，也就是最后一个订阅该事件的处理程序返回的值。这个功能在某些情况下是有用的，但最好使用 void 类型的事件处理程序，且避免使用 out 类型的参数(如果使用 out 参数，参数返回的值的源头就是不清楚的)。

## 4. 匿名方法

除了定义事件处理方法之外，还可以使用匿名方法(anonymous method)。匿名方法实际上并非传统意义上的方法，它不是某个类上的方法，而纯粹是为用作委托目的而创建的。

要创建匿名方法，需要使用下面的代码：

```
delegate(parameters)
{
    // Anonymous method code.
};
```

其中 parameters 是一个参数列表，这些参数匹配正在实例化的委托类型，由匿名方法的代码使用，例如：

```
delegate(Connection source, MessageArrivedEventArgs e)
{
    // Anonymous method code matching MessageHandler event in Ch13Ex03.
};
```

使用这段代码可以完全绕过 Ch13Ex03 中的 DisplayMessage()方法:

```
myConnection1.MessageArrived +=
    delegate(Connection source, MessageArrivedEventArgs e)
    {
        Console.WriteLine("Message arrived from: {0}", source.Name);
        Console.WriteLine("Message Text: {0}", e.Message);
    };
```

对于匿名方法要注意,对于包含它们的代码块来说,它们是局部的,可以访问这个区域内的局部变量。如果使用这样一个变量,它就成为外部变量(outer variable)。外部变量在超出作用域时,是不会删除的,这与其他局部变量不同,在使用它们的匿名方法被销毁时,外部变量才会删除。这比我们希望的时间晚一些,所以要格外小心。如果外部变量占用了大量内存,或者使用的资源在其他方面是比较昂贵的(例如资源在数量上是有限的),就可能导致内存或性能问题。

## 13.4 扩展和使用 CardLib

前面介绍了事件的定义和使用,现在就可以在 Ch13CardLib 中使用它们了。当使用 GetCard 获得 Deck 对象中的最后一个 Card 对象时,就将引发的事件 LastCardDrawn 添加到该类库中。这个事件允许订阅者(subscriber)自动重新洗牌,停止客户要求处理。为这个事件定义的委托(LastCardDrawnHandler)需要为 Deck 对象提供一个引用,这样无论处理程序在什么地方,都可以访问 Shuffle()方法。在 Deck.cs 中添加以下代码:



可从  
wrox.com  
下载源代码

```
namespace Ch13CardLib
{
    public delegate void LastCardDrawnHandler(Deck currentDeck);
```

代码段 Ch13CardLib\Deck.cs

定义和引发事件的代码比较简单,如下所示:

```
public event LastCardDrawnHandler LastCardDrawn;

...

public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
    {
        if ((cardNum == 51) && (LastCardDrawn != null))
            LastCardDrawn(this);
        return cards[cardNum];
    }
    else
```

```

        throw new CardOutOfRangeException((Cards)cards.Clone());
    }

```

这是把事件添加到 Deck 类定义所需要的所有代码。

## CardLib 的扑克牌游戏客户程序

在开发了 CardLib 库后, 就可以使用它了。在结束讲述 C# 和 .NET Framework 中 OOP 技术的这个部分前, 我们将编写扑克牌应用程序的基本代码, 其中将使用我们熟悉的扑克牌类。

与前面的章节一样, 我们将在 Ch13CardLib 解决方案中添加一个客户控制台应用程序, 添加一个 Ch13CardLib 项目的引用, 使之成为启动项目。这个应用程序叫作 Ch13CardClient。

首先在 Ch13CardClient 的一个新文件 Player.cs 中创建一个新类 Player, 这个类包含两个自动属性: Name(字符串) 和 PlayHand(Cards 类型)。这些属性有私有的 get 访问器。但是 PlayHand 属性仍可以对其内容进行写入访问, 这样就可以修改玩家手中的扑克牌。

我们还把默认的构造函数设置为私有, 以隐藏它, 并提供了一个公共的非默认构造函数, 该函数接受 Player 实例中属性 Name 的初始值。

最后, 提供一个 bool 类型的方法 HasWon()。如果玩家手中的扑克牌花色都相同(一个简单的取胜条件, 但并没有什么意义), 该方法就返回 true。

Player.cs 的代码如下所示:



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
    public class Player
    {
        public string Name { get; private set; }

        public Cards PlayHand { get; private set; }

        private Player()
        {
        }

        public Player(string name)
        {
            Name = name;
            PlayHand = new Cards();
        }

        public bool HasWon()
        {
            bool won = true;
            Suit match = PlayHand[0].suit;
            for (int i = 1; i < PlayHand.Count; i++)
            {
                won &= PlayHand[i].suit == match;
            }
        }
    }
}

```

```

    }
    return won;
}
}
}

```

代码段 Ch13CardClient\Player.cs

接着定义一个处理扑克牌游戏的类 `Game`，这个类在 `Ch13CardClient` 项目的 `Game.cs` 文件中。这个类有 4 个私有成员字段：

- `playDeck`——`Deck` 类型的变量，包含要使用的一副扑克牌
- `currentCard`——一个 `int` 值，用作下一张要翻开的扑克牌的指针
- `players`——一个 `Player` 对象数组，表示玩家
- `discardedCards`——`Cards` 集合，表示玩家扔掉的扑克牌，但还没有放回整副牌中。

这个类的默认构造函数初始化了存储在 `playDeck` 中的 `Deck`，并洗牌，把 `currentCard` 指针变量设置为 0(`playDeck` 中的第一张牌)，并关联了 `playDeck.LastCardDrawn` 事件的处理程序 `Reshuffle()`。这个处理程序将洗牌，初始化 `discardedCards` 集合，并把 `currentCard` 重置为 0，准备从新的一副牌中读取扑克牌。

`Game` 类还包含两个实用方法：`SetPlayers()` 可以设置游戏的玩家(`Player` 对象数组)，`DealHands()` 可以处理玩家手中的牌(每个玩家有 7 张牌)。玩家的数量限制为 2~7 人，确保每个玩家有足够多的牌。

最后，`PlayGame()` 方法包含游戏逻辑。在分析了 `Program.cs` 中的代码后介绍这个方法，`Game.cs` 的剩余代码如下所示：



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
    public class Game
    {
        private int currentCard;
        private Deck playDeck;
        private Player[] players;
        private Cards discardedCards;

        public Game()
        {
            currentCard = 0;
            playDeck = new Deck(true);
            playDeck.LastCardDrawn += new LastCardDrawnHandler(Reshuffle);
            playDeck.Shuffle();
            discardedCards = new Cards();
        }

        private void Reshuffle(Deck currentDeck)

```

```

    {
        Console.WriteLine("Discarded cards reshuffled into deck.");
        currentDeck.Shuffle();
        discardedCards.Clear();
        currentCard = 0;
    }

    public void SetPlayers(Player[] newPlayers)
    {
        if (newPlayers.Length > 7)
            throw new ArgumentException("A maximum of 7 players may play this" +
                                         " game.");

        if (newPlayers.Length < 2)
            throw new ArgumentException("A minimum of 2 players may play this" +
                                         " game.");

        players = newPlayers;
    }

    private void DealHands()
    {
        for (int p = 0; p < players.Length; p++)
        {
            for (int c = 0; c < 7; c++)
            {
                players[p].PlayHand.Add(playDeck.GetCard(currentCard++));
            }
        }
    }

    public int PlayGame()
    {
        // Code to follow.
    }
}

```

---

代码段 Ch13CardClient\Game.cs

---

Program.cs 包含 Main() 方法，它启动和运行游戏。这个方法执行以下步骤：

- (1) 显示引导画面。
- (2) 提示用户输入 2~7 个玩家。
- (3) 建立一个 Player 对象数组。
- (4) 给每个玩家起个名字，用于初始化数组中的一个 Player 对象。
- (5) 创建一个 Game 对象，使用 SetPlayers() 方法指定玩家。
- (6) 使用 PlayGame() 方法启动游戏。
- (7) PlayGame() 的 int 返回值用于显示一条获胜消息(返回的值是 Player 对象数组中获胜的玩家的索引)。

这个方法的代码(为了清晰起见，加了一些注释)如下所示：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    // Display introduction.
    Console.WriteLine("KarliCards: a new and exciting card game.");
    Console.WriteLine("To win you must have 7 cards of the same suit in" +
        " your hand.");
    Console.WriteLine();

    // Prompt for number of players.
    bool inputOK = false;
    int choice = -1;
    do
    {
        Console.WriteLine("How many players (2-7)?");
        string input = Console.ReadLine();
        try
        {
            // Attempt to convert input into a valid number of players.
            choice = Convert.ToInt32(input);
            if ((choice >= 2) && (choice <= 7))
                inputOK = true;
        }
        catch
        {
            // Ignore failed conversions, just continue prompting.
        }
    } while (inputOK == false);

    // Initialize array of Player objects.
    Player[] players = new Player[choice];

    // Get player names.
    for (int p = 0; p < players.Length; p++)
    {
        Console.WriteLine("Player {0}, enter your name:", p + 1);
        string playerName = Console.ReadLine();
        players[p] = new Player(playerName);
    }

    // Start game.
    Game newGame = new Game();
    newGame.SetPlayers(players);
    int whoWon = newGame.PlayGame();

    // Display winning player.
    Console.WriteLine("{0} has won the game!", players[whoWon].Name);
}
```

代码段 Ch13CardClient\Program.cs

接着看看应用程序的主体 PlayGame()。由于篇幅所限，这里不准备详细讲解这个方法，而只是加注了一些注释，使之更容易理解。实际上，这些代码都不复杂，仅是较多而已。

每个玩家都可以查看手中的牌和桌面上的一张翻开的牌。他们可以拾取这张牌，或者翻开一张



新牌。在拾取一张牌后，玩家必须扔掉一张牌，如果他们拾取了桌面上的那张牌，就必须用另一张牌替换桌面上的那张牌，或者把扔掉的那张牌放在桌面上那张牌的上面(把扔掉的那张牌添加到 discardedCards 集合中)。

在分析这段代码时，一个关键的问题是 Card 对象的处理方式。必须清楚，这些对象定义为引用类型，而不是值类型(使用结构)。给定的 Card 对象似乎同时存在于多个地方，因为引用可以存在于 Deck 对象、Player 对象的 hand 字段、discardedCards 集合和 playCard 对象(桌面上的当前牌)中。这样就很方便地跟踪扑克牌，特别是可以用于从一副牌中拾取一张新牌。如果牌不在任何玩家的手中，也不在 discardedCards 集合中，才能接受该牌。

代码如下所示：

```
public int PlayGame()
{
    // Only play if players exist.
    if (players == null)
        return -1;

    // Deal initial hands.
    DealHands();

    // Initialize game vars, including an initial card to place on the
    // table: playCard.
    bool GameWon = false;
    int currentPlayer;
    Card playCard = playDeck.GetCard(currentCard++);
    discardedCards.Add(playCard);

    // Main game loop, continues until GameWon == true.
    do
    {
        // Loop through players in each game round.
        for (currentPlayer = 0; currentPlayer < players.Length;
            currentPlayer++)
        {
            // Write out current player, player hand, and the card on the
            // table.
            Console.WriteLine("{0}'s turn.", players[currentPlayer].Name);
            Console.WriteLine("Current hand:");
            foreach (Card card in players[currentPlayer].PlayHand)
            {
                Console.WriteLine(card);
            }
            Console.WriteLine("Card in play: {0}", playCard);

            // Prompt player to pick up card on table or draw a new one.
            bool inputOK = false;
            do
            {
                Console.WriteLine("Press T to take card in play or D to " +
```

```

        "draw:");
string input = Console.ReadLine();
if (input.ToLower() == "t")
{
    // Add card from table to player hand.
    Console.WriteLine("Drawn: {0}", playCard);
    // Remove from discarded cards if possible (if deck
    // is reshuffled it won't be there any more)
    if (discardedCards.Contains(playCard))
    {
        discardedCards.Remove(playCard);
    }
    players[currentPlayer].PlayHand.Add(playCard);
    inputOK = true;
}
if (input.ToLower() == "d")
{
    // Add new card from deck to player hand.
    Card newCard;
    // Only add card if it isn't already in a player hand
    // or in the discard pile
    bool cardIsAvailable;
    do
    {
        newCard = playDeck.GetCard(currentCard++);
        // Check if card is in discard pile
        cardIsAvailable = !discardedCards.Contains(newCard);
        if (cardIsAvailable)
        {
            // Loop through all player hands to see if newCard is
            // already in a hand.
            foreach (Player testPlayer in players)
            {
                if (testPlayer.PlayHand.Contains(newCard))
                {
                    cardIsAvailable = false;
                    break;
                }
            }
        }
    } while (!cardIsAvailable);
    // Add the card found to player hand.
    Console.WriteLine("Drawn: {0}", newCard);
    players[currentPlayer].PlayHand.Add(newCard);
    inputOK = true;
}
} while (inputOK == false);

// Display new hand with cards numbered.
Console.WriteLine("New hand:");

```

```

    for (int i = 0; i < players[currentPlayer].PlayHand.Count; i++)
    {
        Console.WriteLine("{0}: {1}", i + 1,
                           players[currentPlayer].PlayHand[i]);
    }

    // Prompt player for a card to discard.
    inputOK = false;
    int choice = -1;
    do
    {
        Console.WriteLine("Choose card to discard:");
        string input = Console.ReadLine();
        try
        {
            // Attempt to convert input into a valid card number.
            choice = Convert.ToInt32(input);
            if ((choice > 0) && (choice <= 8))
                inputOK = true;
        }
        catch
        {
            // Ignore failed conversions, just continue prompting.
        }
    } while (inputOK == false);

    // Place reference to removed card in playCard (place the card
    // on the table), then remove card from player hand and add
    // to discarded card pile.
    playCard = players[currentPlayer].PlayHand[choice - 1];
    players[currentPlayer].PlayHand.RemoveAt(choice - 1);
    discardedCards.Add(playCard);
    Console.WriteLine("Discarding: {0}", playCard);

    // Space out text for players
    Console.WriteLine();

    // Check to see if player has won the game, and exit the player
    // loop if so.
    GameWon = players[currentPlayer].HasWon();
    if (GameWon == true)
        break;
}
} while (GameWon == false);

// End game, noting the winning player.
return currentPlayer;
}

```

图 13-8 显示了一个正在进行的比赛。

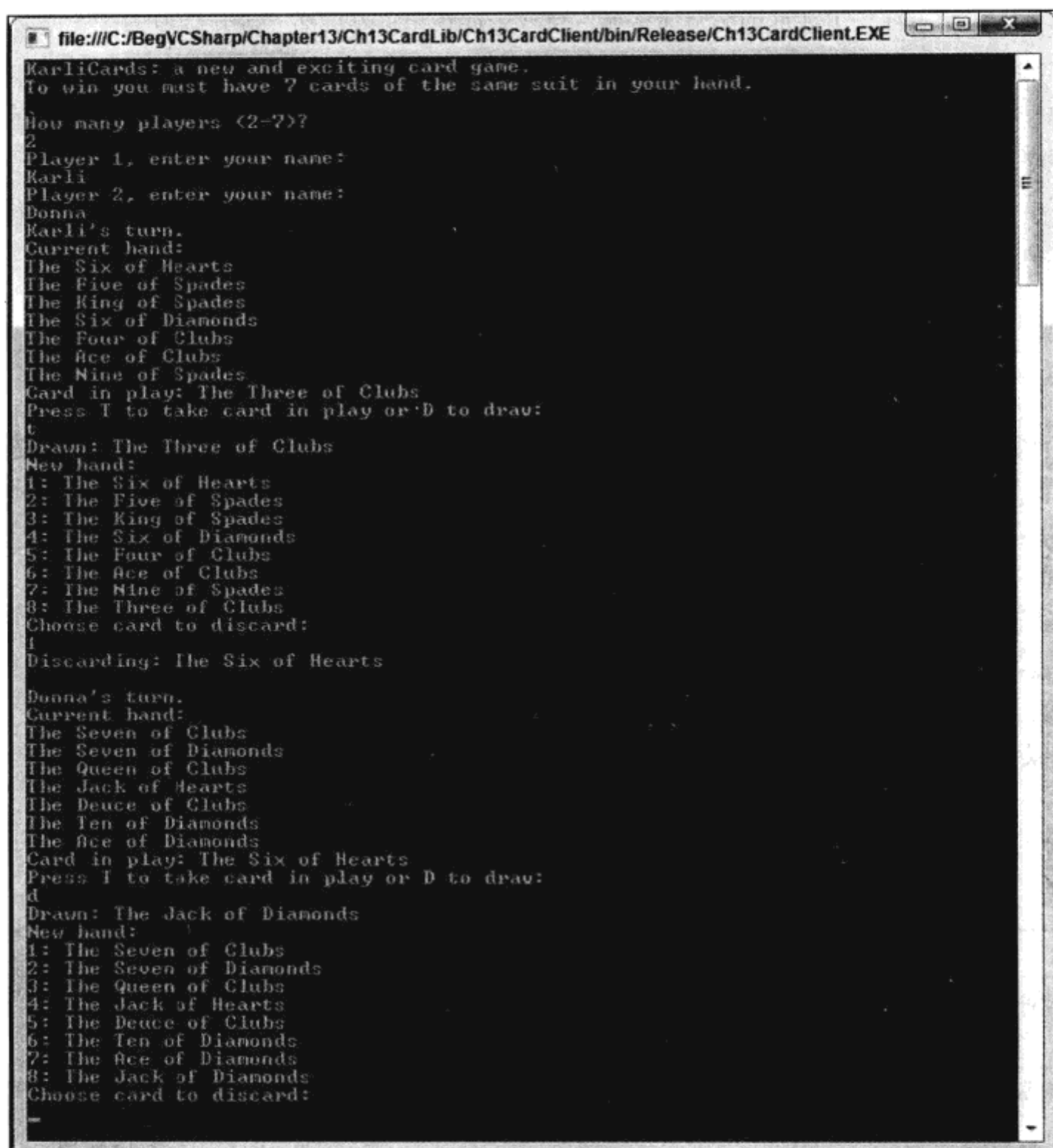


图 13-8

玩这个游戏，确保花一些时间去仔细研究它。应尝试在 `Reshuffle()` 方法中设置一个断点，由 7 个玩家来玩这个游戏。如果在拾取了扑克牌后，马上扔掉它，不需要太长的时间就要重新洗牌了，因为 7 个玩家玩这个游戏时，就只富余 3 张牌。这样就可以注意 3 张牌何时重新翻开，验证程序是否正常执行。

## 13.5 小结

本章介绍了一些高级技术，扩展讨论了 C# 语言的基础知识。首先介绍了名称空间的限定、`::` 运算符和 `global` 关键字，来确保对类型的引用是希望的类型引用。接着讨论了如何实现自己的异常对象，将更详细的信息传送给异常处理程序。然后在前几章开发的扑克牌游戏项目 `CardLib` 的代码中使用定制异常。

最后论述了事件和事件处理的重要论题。尽管事件是相当微妙的，刚开始很难理解，但它的代码是非常简单的，读者肯定会在本书的其他地方经常使用事件处理程序。我们还讨论了事件和事件处理的一些简单示例，修改了 CardLib 类库，使用这个类库创建了一个简单的客户扑克牌游戏程序。这个程序可以作为本书到现在为止介绍的所有技术的一个实用示例。

本章不仅完成了 OOP 作为 C#编程技术的讨论，还完成了C#语言的讨论。第 14 章将介绍 C# 3.0 和 4 中新增的特性。

13.6 练习

- (1) 编写事件处理程序的代码，这些代码使用了通用语法(object sender, EventArgs e)，该语法将接受本章前面的 Timer.Elapsed 或 Connection.MessageArrived 事件。处理程序应输出一个表示接收了什么类型事件的字符串，并根据引发的事件，输出 MessageArrivedEventArgs 参数的 Message 属性或 ElapsedEventArgs 参数的 SignalTime 属性。
- (2) 修改扑克牌游戏示例，设置流行拉米扑克牌的更有趣的取胜条件。即一个玩家要取胜，手中的牌必须包含两套牌，一套由 3 张牌组成，另一套由 4 张牌组成。一套牌应是连续的同花色的牌(例如，3H、4H、5H、6H)或者几张同点的牌(例如，2H、2D、2S)。

附录 A 给出了练习答案。

13.7 本章要点

主 题	重 要 概 念
名称空间限定符	为了避免名称空间限定的模糊性，可以使用::运算符强制编译器使用已创建好的别名。还可以使用 global 命名空间作为顶级名称空间的别名
定制异常	从根类 Exception 中派生，就可以创建自己的异常类。这是有益的，因为可以更多地控制特定异常的捕获，并允许定制包含在异常中的数据，以高效地处理它
事件处理	许多类都提供了事件，在代码中发生某个触发器时，就会引发这些事件。可以为这些事件编写处理程序，在引发事件时执行代码。这种双向的通信方式是响应代码的一种良好机制，无需编写可能导致改变对象的复杂、费解的代码
事件定义	可以定义自己的异常类型，这涉及给事件的处理程序创建指定的事件和委托类型。可以使用标准的、无返回类型的委托类型和派生于 System.EventArgs 的定制事件参数，使事件处理程序有多种用途。还可以使用 EventHandler 和 EventHandler<T>委托类型，以便通过更简单的代码定义事件
匿名方法	为了使代码更便于阅读，常常可以使用匿名方法来替代完整的事件处理方法。这表示，在添加事件处理程序的地方直接定义要在引发事件时执行的代码，为此需要使用 delegate 关键字

# 第 14 章

## C#语言的改进

### 本章内容:

---

- 如何使用初始化器
- var 类型是什么, 如何使用类型推理
- 如何使用匿名类型
- dynamic 类型是什么, 如何使用它
- 如何使用命名和可选的方法参数
- 如何使用扩展方法
- Lambda 表达式是什么, 如何使用它们

C#语言不是一成不变的, C#的发明者 Anders Hejlsberg 和微软公司的其他人一直在更新和改进该语言。在编写本书时, 最新的改进都放在 C#语言的第 4 版本中, 它作为 VS2010 系列产品的一部分发布。阅读了本书前面的内容后, 读者可能会考虑还需要什么其他功能。实际上, C#以前的版本从功能的角度来看并不缺乏什么, 但这并不意味着无法进一步简化 C#编程的某些方面, 或者 C#和其他技术之间的关系不能更加流畅。

理解这点的最佳方式是考虑该语言的 1.0 和 2.0 版本之间新增的内容——泛型。泛型虽然非常有用, 但并没有真正提供以前不能实现的功能。的确, 泛型大大简化了编程, 但没有它们, 就需要编写更多的代码。我们都不想回到以前没有泛型集合类的日子。无论如何, 泛型并不是 C#的基础部分, 只是该语言的改进。

C#语言后续的改进也是这样, 它们为以前不借助冗长和/或高级编程技术时很难实现的功能提供了新的方式。本章将介绍其中的几处改进, 一些改进(例如变体)已经在本章的对应章节做了介绍。

### 14.1 初始化器

前面的章节学习了如何用各种方式实例化和初始化对象。它们都需要在类定义中添加额外代码, 以便使用独立的语句来初始化或实例化对象。我们还了解了如何创建各种类型的集合类, 包括泛型

集合类。另外，把集合的创建和在集合中添加数据项合并起来并没有什么简便的方法。

对象初始化器提供了一种简化代码的方式，可以合并对象的实例化和初始化。集合初始化器提供了一种简洁的语法，您使用一个步骤就可以创建和填充集合。本节就介绍如何使用这两个新特性。

### 14.1.1 对象初始化器

考虑下面的简单类定义：

```
public class Curry
{
    public string MainIngredient {get; set;}
    public string Style {get; set;}
    public int Spiciness {get; set;}
}
```

这个类有 3 个属性，用第 10 章介绍的自动属性语法来定义。如果希望实例化和初始化这个类的一个对象实例，就必须执行如下几个语句：

```
Curry tastyCurry = new Curry();
tastyCurry.MainIngredient = "panir tikka";
tastyCurry.Style = "jalfrezi";
tastyCurry.Spiciness = 8;
```

如果类定义中未包含构造函数，这段代码就使用 C# 编译器提供的默认无参数构造函数。为了简化这个初始化过程，可以提供一个合适的非默认构造函数：

```
public class Curry
{
    public Curry(string mainIngredient, string style,
                int spiciness)
    {
        MainIngredient = mainIngredient;
        Style = style;
        Spiciness = spiciness;
    }
    ...
}
```

这样就可以编写代码，把实例化和初始化合并起来：

```
Curry tastyCurry = new Curry("panir tikka", "jalfrezi", 8);
```

这段代码工作得很好，但它会强制使用 Curry 类的代码使用这个构造函数，这将阻止前面使用无参构造函数的代码运行。常常需要提供无参构造函数，在必须序列化类时，情况尤其如此：

```
public class Curry
{
    public Curry()
    {
    }
    ...
}
```



现在可以用任意方式实例化和初始化 Curry 类，但已在最初的类定义中添加几行代码，与提供基本的执行代码相比，这种方法并没有做更多的工作。

进入对象初始化器(object initializer)，这是无需在类中添加额外的代码(如此处详细说明了构造函数)就可以实例化和初始化对象的方式。实例化对象时，要为每个需初始化的、可公开访问的属性或字段使用名称-值对，来提供其值。其语法如下：

```
<className> <variableName> = new <className>
{
    <propertyOrField1> = <value1>,
    <propertyOrField2> = <value2>,
    ...
    <propertyOrFieldN> = <valueN>
};
```

例如，重写前面的代码，实例化和初始化 Curry 类型的一个对象，如下所示：

```
Curry tastyCurry = new Curry
{
    MainIngredient = "panir tikka",
    Style = "jalfrezi",
    Spiciness = 8
};
```

我们常常可以把这样的代码放在一行上，而不会严重影响可读性。

使用对象初始化器时，不必显式调用类的构造函数。如果像上述代码那样省略构造函数的括号，就自动调用默认的空参构造函数。这是在初始化器设置参数值之前调用的，以便在需要时为默认构造函数中的参数提供默认值。另外，可以调用特定的构造函数。同样，先调用这个构造函数，所以在构造函数中对公共属性进行的初始化可能会被初始化器中提供的值覆盖。必须按顺序访问所使用的构造函数(如果没有显式指出，就执行默认的构造函数)，对象初始化器才能正常工作。

如果要用对象初始化器进行初始化的属性比本例中使用的简单类型还复杂，可以使用嵌套的对象初始化器，即使用与前面相同的语法：

```
Curry tastyCurry = new Curry
{
    MainIngredient = "panir tikka",
    Style = "jalfrezi",
    Spiciness = 8,
    Origin = new Restaurant
    {
        Name = "King's Balti",
        Location = "York Road",
        Rating = 5
    }
};
```

这里初始化了一个 Restaurant 类型(这里没有列出)的 Origin 属性。代码初始化了 Origin 属性的 3 个特性：Name、Location 和 Rating，其值的类型分别是 string、string 和 int 类型。这个初始化使用了嵌套的对象初始化器。

注意，对象初始化器没有替代非默认的构造函数。在初始化对象时，可以使用对象初始化器来

设置属性和字段值，但这并不意味着总是知道需要初始化什么状态。通过构造函数，可以准确地指定对象需要什么值才能起作用，再执行代码，以立即响应这些值。

### 14.1.2 集合初始化器

第 5 章描述了如何使用如下语法，用值来初始化数组：

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

这是一种合并实例化和初始化数组的简捷方式。集合初始化器只是把这个语法扩展到集合上：

```
List < int > myIntCollection = new List < int > {5, 9, 10, 2, 99};
```

通过合并对象和集合初始化器，就可以用简洁的代码配置集合了。下面的代码：

```
List < Curry > curries = new List < Curry > ();
curries.Add(new Curry("Chicken", "Pathia", 6));
curries.Add(new Curry("Vegetable", "Korma", 3));
curries.Add(new Curry("Prawn", "Vindaloo", 9));
```

可以用如下代码替换：

```
List < Curry > moreCurries = new List < Curry >
{
    new Curry
    {
        MainIngredient = "Chicken",
        Style = "Pathia",
        Spiciness = 6
    },
    new Curry
    {
        MainIngredient = "Vegetable",
        Style = "Korma",
        Spiciness = 3
    },
    new Curry
    {
        MainIngredient = "Prawn",
        Style = "Vindaloo",
        Spiciness = 9
    }
};
```

这非常适合于主要用于数据表示的类型，而且，集合初始化和本书后面介绍的 LINQ 技术一起使用时效果极佳。

下面的示例说明了如何使用对象和集合初始化器。

#### 试一试：使用初始化器

- (1) 创建一个新的控制台应用程序 Ch14Ex01，把它保存在 C:\BegVCSharp\Chapter14 目录下。
- (2) 在 Solution Explorer 窗口中右击项目名称，选择 Add | Existing Item 选项。
- (3) 在 C:\BegVCSharp\Chapter12\Ch12Ex04\Ch12Ex04 目录中选择 Animal.cs、Cow.cs、Chicken.cs、

SuperCow.cs 和 Farm.cs 文件，单击 Add 按钮。

(4) 修改文件中已添加的名称空间声明，如下所示：

```
namespace Ch14Ex01
```

(5) 删除 Cow、Chicken 和 SuperCow 类的构造函数。

(6) 修改 Program.cs 中的代码，如下所示：



```
static void Main(string[] args)
{
    Farm < Animal > farm = new Farm < Animal >
    {
        new Cow { Name="Norris" },
        new Chicken { Name=" Rita" },
        new Chicken(),
        new SuperCow { Name="Chesney" }
    };
    farm.MakeNoises();
    Console.ReadKey();
}
```

代码段 Ch14Ex01\Program.cs

(7) 生成应用程序，会得到如图 14-1 所示的生成错误。

Error List					
4 Errors 0 Warnings 0 Messages					
	Description	File	Line	Column	Project
* 1	'Ch14Ex01.Farm<Ch14Ex01.Animal>' does not contain a definition for 'Add'	Program.cs	14	9	Ch14Ex01
* 2	'Ch14Ex01.Farm<Ch14Ex01.Animal>' does not contain a definition for 'Add'	Program.cs	15	9	Ch14Ex01
* 3	'Ch14Ex01.Farm<Ch14Ex01.Animal>' does not contain a definition for 'Add'	Program.cs	16	9	Ch14Ex01
* 4	'Ch14Ex01.Farm<Ch14Ex01.Animal>' does not contain a definition for 'Add'	Program.cs	17	9	Ch14Ex01

图 14-1

(8) 给 Farm.cs 添加如下代码：



```
public class Farm<T> : IEnumerable<T>
    where T : Animal
{
    public void Add(T animal)
    {
        animals.Add(animal);
    }

    ...
}
```

代码段 Ch14Ex01\Farm.cs

(9) 运行应用程序，结果如图 14-2 所示。

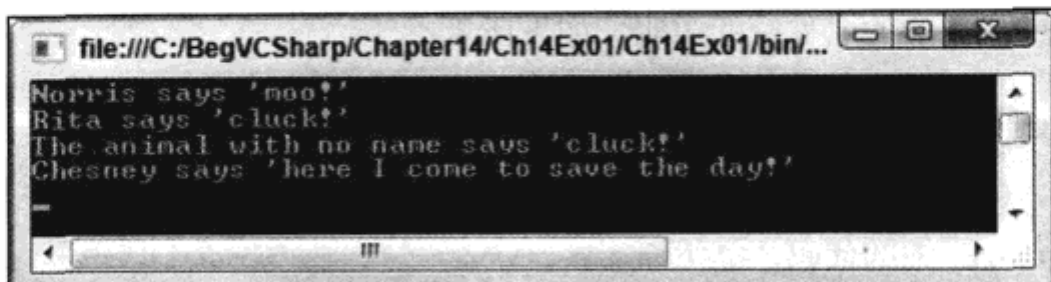


图 14-2

### 示例的说明

这个示例合并了对象和集合初始化器，用一个步骤创建并填充了一个对象集合。它使用了前面章节介绍的 `farmyard` 对象集合，但需要对用于这些类的初始化器进行两处修改。

首先，给派生于 `Animal` 基类的类删除构造函数。可以删除这些构造函数，是因为它们设置了动物的 `Name` 属性，而这里使用对象初始化器来完成。另外，还可以添加默认的构造函数。无论采用哪种方式，在使用默认的构造函数时，会根据基类中的默认构造函数来初始化 `Name` 属性，代码如下：

```
public Animal()
{
    name = "The animal with no name";
}
```

但是，对象初始化器与派生于 `Animal` 类的类一起使用时，初始化器设置的任何属性都在对象实例化后设置，因此在执行这个基类的构造函数之后设置。如果 `Name` 属性的值作为对象初始化器的一部分提供，这个值就会覆盖默认值。在示例代码中，为添加到集合中的所有项设置了 `Name` 属性，只有一项除外。

其次，必须给 `Farm` 类添加 `Add()` 方法，否则会响应如下形式的一系列编译错误：

```
'Ch14Ex01.Farm < Ch14Ex01.Animal >' does not contain a definition for 'Add'
```

这个错误显示出了集合初始化器的部分底层功能。在后台，编译器为在集合初始化器中提供的每一项调用集合的 `Add()` 方法。`Farm` 类通过 `Animals` 属性提供了一个 `Animal` 对象集合。编译器猜不出这就是要(通过 `Animals.Add()`)填充的属性，所以代码会失败。为了更正这个问题，可以把 `Add()` 方法添加到类中，类通过对象初始化器进行初始化。

还可以修改示例中的代码，为 `Animals` 属性提供一个嵌套的初始化器，如下所示：

```
static void Main(string[] args)
{
    Farm < Animal > farm = new Farm < Animal >
    {
        Animals =
        {
            new Cow { Name="Norris" },
            new Chicken { Name="Rita" },
            new Chicken(),
            new SuperCow { Name="Chesney" }
        }
    }
}
```

```
};
farm.MakeNoises();
Console.ReadKey();
}
```

有了此代码，就不需要为 Farm 类提供 Add()方法了。这个备用技巧适用于包含多个集合的类。在这种情况下，用包含类的 Add()方法添加的集合没有其他方式。

## 14.2 类型推理

本书前面介绍过 C#是一种强类型化的语言，这表示每个变量都有固定的类型，只能用于接受该类型的代码中。在前面的所有代码示例中，都用如下形式的代码来声明变量：

```
<type> <varName> ;
```

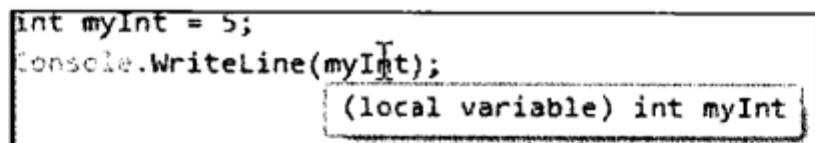
或者

```
<type> <varName> = <value> ;
```

下面的代码显示了变量 varName 的类型：

```
int myInt = 5;
Console.WriteLine(myInt);
```

把鼠标指针停放在变量标识符上，IDE 就会显示该变量的类型，如图 14-3 所示。



```
int myInt = 5;
Console.WriteLine(myInt);
(local variable) int myInt
```

图 14-3

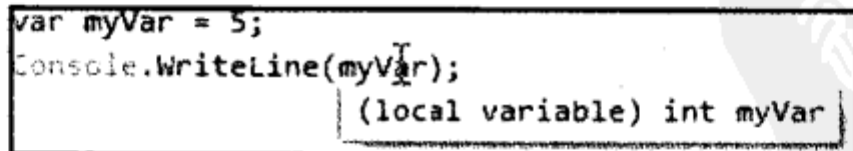
C# 3.0 引入了新关键字 var，它可以替代前面代码中的 type：

```
var <varName> = <value> ;
```

在这行代码中，变量<varName>隐式地类型化为 value 的类型。注意，类型的名称并不是 var。在下面的代码中：

```
var myVar = 5;
```

myVar 是 int 类型的变量，而不是 var 类型的变量，如图 14-4 所示，IDE 也显示了其类型。



```
var myVar = 5;
Console.WriteLine(myVar);
(local variable) int myVar
```

图 14-4

这是非常重要的一点。使用 var 时，并不是声明了一个没有类型的变量，也不是声明了一个类型可以变化的变量。否则，C#就不再是强类型化的语言了。我们只需利用编译器确定变量的类型即可。



.NET 4 引入的动态类型扩展了 C# 是强类型化语言的定义，参见本章后面的“动态查找”一节。

如果编译器不能确定用 `var` 声明的变量类型，代码就不会编译。因此，在用 `var` 声明变量时，必须同时初始化该变量，因为如果没有初始值，编译器就不能确定变量的类型。下面的代码就不能编译：

```
var myVar;
```

`var` 关键字还可以通过数组初始化器来推断数组的类型：

```
var myArray = new[] {4, 5, 2};
```

在这行代码中，`myArray` 的类型被隐式地设置为 `int[]`。在用这种方式隐式指定数组的类型时，初始化器中使用的数组元素必须是以下情形中的一种：

- 相同的类型
- 相同的引用类型或空
- 所有元素的类型都可以隐式地转换为一个类型

如果应用最后一条规则，元素可以转换的类型就称为数组元素的最佳类型。如果这个最佳类型有任何含糊的地方，即所有元素的类型都可以隐式转换为两种或更多的类型，代码就不会编译。我们会接收到错误，错误中指出没有最佳类型：

```
var myArray = new[] {4, "not an int", 2};
```

还要注意数字值从来都不会解释为可空类型，所以下面的代码无法编译：

```
var myArray = new[] {4, null, 2};
```

但可以使用标准的数组初始化器，使如下代码编译：

```
var myArray = new int?[] {4, null, 2};
```

最后一点要说明的是，标识符 `var` 并非不能用于类名。这意味着，如果代码在其作用域中(在同一个名称空间或引用的名称空间中)有一个 `var` 类，就不能使用 `var` 关键字的隐式类型化功能。

类型推理功能本身并不是很有效，因为在本节前面的代码中，它只会使事情更复杂。使用 `var` 会加大判断给定变量的类型的难度。但是如本章后面所述，推断类型的概念非常重要，因为它是其他技术的基础。下一个主题是匿名类型，它就以推断类型为基础。

## 14.3 匿名类型

在编写程序一段时间后，会发现我们要花很多时间为数据表示创建简单、乏味的类，在数据库应用程序中尤其如此。常常有一系列类只提供属性。本章前面的 `Curry` 类就是一个很好的例子：

```
public class Curry
```

```

{
    public string MainIngredient {get; set;}
    public string Style {get; set;}
    public int Spiciness {get; set;}
}

```

这个类什么也没做，只是存储结构化数据。在数据库或电子表格中，可以把这个类看作表中的一行。可以保存这个类的实例的集合类应表示表或电子表格中的多个行。

这是类完全可以接受的一种用法，但编写这些类的代码比较单调，对底层数据模式的任何修改都需要添加、删除或修改定义类的代码。

匿名类型(**anonymous type**)是简化这个编程模型的一种方式。其理念是使用 C#编译器根据要存储的数据自动创建类型，而不是定义简单的数据存储类型。

可以按如下方式实例化前面的 Curry 类型：

```

Curry curry = new Curry
{
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
};

```

也可以使用匿名类型，如下所示：

```

var curry = new
{
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
};

```

这里有两个区别。第一，使用了 **var** 关键字。这是因为匿名类型没有可以使用的标识符。稍后可以看到，它们在内部有一个标识符，但不能在代码中使用。第二，在 **new** 关键字的后面没有指定类型名称，这是编译器确定我们要使用匿名类型的方式。

IDE 检测到匿名类型定义后，就会相应地更新 **IntelliSense**。通过前面的声明，可以看到如图 14-5 所示的匿名类型。

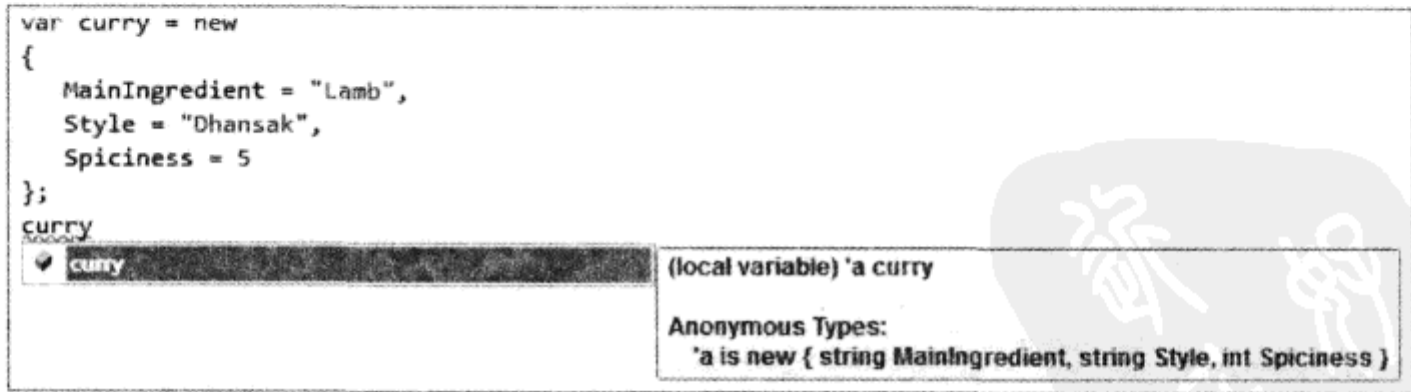


图 14-5

其中，变量 **curry** 的类型是 **'a'**。显然，不能在代码中使用这个类型——它甚至不是合法的标识符名称。'符号仅用于在 **IntelliSense** 中表示匿名类型。**IntelliSense** 也允许查看匿名类型的成员，如图 14-6 所示。



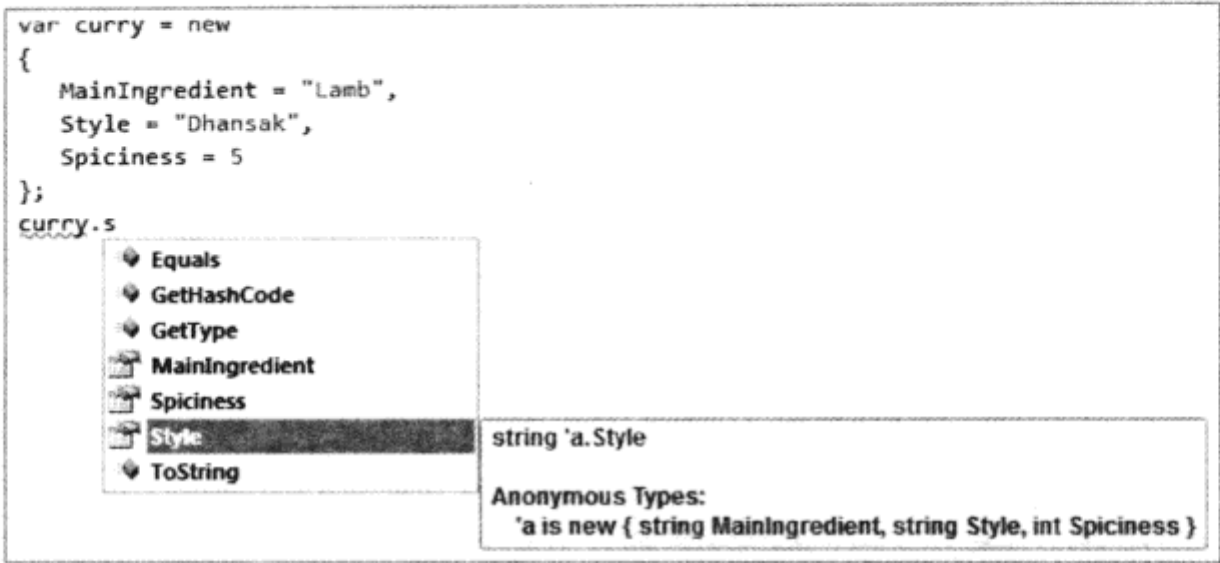


图 14-6

注意，这里显示的属性定义为只读属性。这表示，如果要在数据存储对象中修改属性的值，就不能使用匿名类型。

还实现了匿名类型的其他成员，如下面的示例所示。

试一试：使用匿名类型

- (1) 创建一个新的控制台应用程序 Ch14Ex02，把它保存在 C:\BegVCSharp\Chapter14 目录中。
- (2) 修改 Program.cs 中的代码，如下所示：



```
static void Main(string[] args)
{
    var curries = new[]
    {
        new
        {
            MainIngredient = "Lamb",
            Style = "Dhansak",
            Spiciness = 5
        },
        new
        {
            MainIngredient = "Lamb",
            Style = "Dhansak",
            Spiciness = 5
        },
        new
        {
            MainIngredient = "Chicken",
            Style = "Dhansak",
            Spiciness = 5
        }
    };
    Console.WriteLine(curries[0].ToString());
    Console.WriteLine(curries[0].GetHashCode());
    Console.WriteLine(curries[1].GetHashCode());
    Console.WriteLine(curries[2].GetHashCode());
    Console.WriteLine(curries[0].Equals(curries[1]));
}
```

```

        Console.WriteLine(curries[0].Equals(curries[2]));
        Console.WriteLine(curries[0] == curries[1]);
        Console.WriteLine(curries[0] == curries[2]);

        Console.ReadKey();
    }
}

```

代码段 Ch14Ex02\Program.cs

(3) 运行应用程序，结果如图 14-7 所示。



图 14-7

### 示例的说明

这个示例创建了一个匿名类型对象的数组，然后使用它测试匿名类型提供的成员。创建匿名类型对象的数组的代码如下：

```

var curries = new[]
{
    new
    {
        MainIngredient = "Lamb",
        Style = "Dhansak",
        Spiciness = 5
    },
    ...
};

```

这段代码通过本节和前面“类型推理”一节中介绍的语法，使用了隐式类型化为匿名类型的数组。结果是 `curries` 变量包含匿名类型的 3 个实例。

在创建了这个数组后，代码首先输出在匿名类型上调用 `ToString()` 的结果：

```
Console.WriteLine(curries[0].ToString());
```

输出结果如下：

```
{ MainIngredient = Lamb, Style = Dhansak, Spiciness = 5 }
```

匿名类型上的 `ToString()` 的实现输出了为该类型定义的每个属性的值。

接着，代码在数组的 3 个对象上分别调用 `GetHashCode()`：

```

Console.WriteLine(curries[0].GetHashCode());
Console.WriteLine(curries[1].GetHashCode());
Console.WriteLine(curries[2].GetHashCode());

```

GetHashCode()执行时,应根据对象的状态为对象返回一个唯一的整数。数组中的前两个对象有相同的属性值,所以其状态是相同的。这些调用的结果是前两个对象的整数相同,第三个对象的整数不同。结果如下:

```
294897435
294897435
621671265
```

接着,调用 Equals()方法比较第一个对象和第二个对象,再比较第一个对象和第三个对象:

```
Console.WriteLine(curries[0].Equals(curries[1]));
Console.WriteLine(curries[0].Equals(curries[2]));
```

结果如下:

```
True
False
```

匿名类型上的Equals()的实现比较对象的状态,如果一个对象的每个属性值都与另一个对象的对应属性值相同,结果就是 true。

但使用==运算符不会得到这样的结果。如前几章所述,==运算符比较对象引用。最后一部分代码进行与上一段代码相同的比较,但用==替代了Equals()方法:

```
Console.WriteLine(curries[0] == curries[1]);
Console.WriteLine(curries[0] == curries[2]);
```

Curries 数组中的每一项都引用匿名类型的不同实例,所以在两种情况下结果都是 false。输出结果与预期的相同:

```
False
False
```

有趣的是,在创建匿名类型的实例时,编译器会注意到,参数是相同的,所以创建同一个匿名类型的 3 个实例——而不是 3 个不同的匿名类型。但是,这并不意味着实例化匿名类型的对象时,编译器会查找匹配的类型。即使在其他地方定义了一个有匹配属性的类,如果使用了匿名类型语法,也只创建(或重用,如本例所示)一个匿名类型。

## 14.4 动态查找

如前所述, var 关键字本身并不是一个类型,所以并没有违反 C#的“强类型化”方法论。但在 C# 4 中做了少许改动。C# 4 引入了“动态变量”的概念,顾名思义,动态变量是类型不固定的变量。

引入动态变量的主要目的是在许多情况下,都希望使用 C#处理另一种语言创建的对象。这包括与旧技术的交互操作,例如 Component Object Model (COM),以及处理动态语言,例如 JavaScript、Python 和 Ruby。在过去,没有非常详细的实现细节,使用 C#访问这些语言所创建的对象的方法和属性,需要用到笨拙的语法。例如,假定代码从 JavaScript 中获得了一个带 Add()方法的对象,该方法把两个数字加在一起。如果没有动态查找功能,调用这个方法的代码就如下所示:

```
ScriptObject jsObj = SomeMethodThatGetsTheObject();
int sum = Convert.ToInt32(jsObj.Invoke("Add", 2, 3));
```

ScriptObject 类型(这里不深入探讨)提供了一种访问 JavaScript 对象的方式,但不能执行如下操作:

```
int sum = jsObj.Add(2, 3);
```

动态查找功能改变了这一切,它允许编写上述代码,但如下面几节所述,这个功能是有代价的。

另一个可以使用动态查找功能的情形是处理未知类型的 C#对象。这听起来似乎很古怪,但这种情形出现的次数比我们想象得多。如果需要编写一些泛型代码,来处理它接收的输入,这也是一个重要的功能。处理这种情形的“旧”方法称为“反射(reflection)”,它涉及使用类型信息来访问类型和成员。实际上,反射的语法非常类似于上述代码中访问 JavaScript 对象的语法,也非常麻烦。

在后台,动态查找功能由 Dynamic Language Runtime (动态语言运行库, DLR)支持。与 CLR 一样,DLR 是 .NET 4 的一部分。DLR 的精确描述及其如何简化交互操作超出了本书的范围,这里仅对如何在 C#中使用它感兴趣。

#### 14.4.1 dynamic 类型

C# 4 引入了 dynamic 关键字,以用于定义变量。例如:

```
dynamic myDynamicVar;
```

与前面介绍的 var 关键字不同,的确存在 dynamic 类型,所以在声明 myDynamicVar 时,无需初始化它的值。



dynamic 类型不同寻常之处是,它仅在编译期间存在,在运行期间它会被 System.Object 类型替代。这是较细微的实现细节,但必须记住这一点,因为这可能澄清了后面的一些讨论。一旦有了动态变量,就可以继续访问其成员(这里没有列出实际获取变量值的代码)。

```
myDynamicVar.DoSomething("With this!");
```

无论 myDynamicVar 实际包含什么值,这行代码都会编译。但是,如果所请求的成员不存在,在执行这行代码时会生成一个 RuntimeBinderException 类型的异常。

实际上,像这样的代码提供了一个应在运行期间应用的“处方”。检查 myDynamicVar 的值,定位带一个字符串参数的 DoSomething()方法,并在需要时调用它。

这最好举例说明。



**警告:** 下面的示例仅用于演示! 一般情况下,应仅在动态类型是唯一的选项时使用它们,例如处理非 .NET 对象。

## 试一试：使用动态类型

- (1) 创建一个新的控制台应用程序 Ch14Ex03，把它保存在 C:\BegVCSharp\Chapter14 目录中。
- (2) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CSharp.RuntimeBinder;
namespace Ch14Ex03
{
    class MyClass1
    {
        public int Add(int var1, int var2)
        {
            return var1 + var2;
        }
    }

    class MyClass2
    {
    }

    class Program
    {
        static int callCount = 0;

        static dynamic GetValue()
        {
            if (callCount++ == 0)
            {
                return new MyClass1();
            }
            return new MyClass2();
        }

        static void Main(string[] args)
        {
            try
            {
                dynamic firstResult = GetValue();
                dynamic secondResult = GetValue();
                Console.WriteLine("firstResult is: {0}",
                    firstResult.ToString());
                Console.WriteLine("secondResult is: {0}",
                    secondResult.ToString());
                Console.WriteLine("firstResult call: {0}",
                    firstResult.Add(2, 3));
                Console.WriteLine("secondResult call: {0}",
                    secondResult.Add(2, 3));
            }
            catch (RuntimeBinderException ex)
            {
            }
        }
    }
}
```

```

        Console.WriteLine(ex.Message);
    }
    Console.ReadKey();
}
}
}

```

代码段 Ch14Ex03\Program.cs

(3) 运行应用程序，结果如图 14-8 所示。

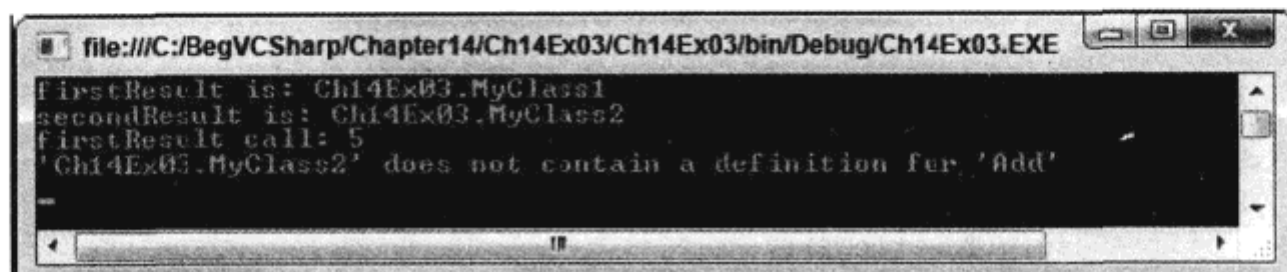


图 14-8

### 示例的说明

这个示例使用一个方法返回两个类型的对象中的一个，以获取动态值，再尝试使用所获取的对象。代码在编译时没有遇到任何问题，但尝试访问不存在的方法时，抛出(并处理)了一个异常。

首先，为包含 `RuntimeBindingException` 异常的名称空间添加一个 `using` 语句：

```
using Microsoft.CSharp.RuntimeBinder;
```

接着定义两个类 `MyClass1` 和 `MyClass2`，其中 `MyClass1` 包含 `Add()` 方法，而 `MyClass2` 不含成员：

```

class MyClass1
{
    public int Add(int var1, int var2)
    {
        return var1 + var2;
    }
}

class MyClass2
{
}

```

还要给 `Program` 类添加一个字段(`callCount`)和一个方法(`GetValue()`)，以获取其中一个类的实例：

```

static int callCount = 0;

static dynamic GetValue()
{
    if (callCount++ == 0)
    {
        return new MyClass1();
    }
}

```

```
    return new MyClass2();
}
```

使用一个简单的调用计数器，这样，第一次调用这个方法时，返回 `MyClass1` 的一个实例，之后返回 `MyClass2` 的实例。注意 `dynamic` 关键字可以用作方法的返回类型。

接着，`Main()` 中的代码调用 `GetValue()` 方法两次，再尝试在返回的两个值上依次调用 `GetString()` 和 `Add()`。这些代码放在 `try ... catch` 块中，以捕获可能发生的 `RuntimeBinderException` 类型的异常。

```
static void Main(string[] args)
{
    try
    {
        dynamic firstResult = GetValue();
        dynamic secondResult = GetValue();
        Console.WriteLine("firstResult is: {0}",
            firstResult.ToString());
        Console.WriteLine("secondResult is: {0}",
            secondResult.ToString());
        Console.WriteLine("firstResult call: {0}",
            firstResult.Add(2, 3));
        Console.WriteLine("secondResult call: {0}",
            secondResult.Add(2, 3));
    }
    catch (RuntimeBinderException ex)
    {
        Console.WriteLine(ex.Message);
    }

    Console.ReadKey();
}
```

可以肯定，调用 `secondResult.Add()` 时会抛出一个异常，因为在 `MyClass2` 上不存在这个方法。异常消息说明了这一点。

`dynamic` 关键字也可以用于其他需要类型名的地方，例如方法参数。`Add()` 方法可以重写为：

```
public int Add(dynamic var1, dynamic var2)
{
    return var1 + var2;
}
```

这对结果没有任何影响。在这个例子中，传送给 `var1` 和 `var2` 的值在运行期间检查，以确定加号+是否存在一个兼容的运算符定义。如果传送了两个 `int` 值，就存在这样的运算符。如果使用了不兼容的值，就抛出 `RuntimeBinderException` 异常。例如，如果尝试：

```
Console.WriteLine("firstResult call: {0}", firstResult.Add("2", 3));
```

异常消息就如下所示：

```
Cannot implicitly convert type 'string' to 'int'
```

从这里获得的教训是动态类型是非常强大的，但有一个警告。如果用强类型代替动态类型，就完全可以避免抛出这些异常。对于大多数自己编写的 C# 代码，应避免使用 `dynamic` 关键字。但是，



如果需要使用它，就应使用它，并会喜欢上它——而不像过去那些可怜的程序员那样没有这个强大的工具可用。

#### 14.4.2 IDynamicMetaObjectProvider

在继续之前，应注意如何使用动态类型，或者更确切地讲，在运行期间对成员访问应用某种技术时会发生什么。实际上，有 3 种不同的方式访问成员：

- 如果动态值是 COM 对象，就使用 COM 技术访问成员(通过 IUnknown 接口，但这里不需要了解它)。
- 如果动态值支持 IDynamicMetaObjectProvider 接口，就使用该接口访问类型成员。
- 如果不能使用上述两种技术，就使用反射。

第二种情形比较有趣，它涉及到 IDynamicMetaObjectProvider 接口。这里不探讨具体的细节，只是注意可以实现这个接口，来准确地控制在运行期间访问成员时会发生什么。但这是高级编程图书的一个主题，因此这里不讨论。

### 14.5 高级方法参数

C# 4 扩展了定义和使用方法参数的方式。这主要是为了响应使用外部定义的接口时出现的一个特殊问题，例如 Microsoft Office 编程模型。其中，一些方法有大量的参数，许多参数并不是每次调用都需要的。过去，这意味着需要一种方式指定缺失的参数，否则在代码中会出现许多空值：

```
RemoteCall(var1, var2, null, null, null, null, null);
```

在这行代码中，null 值表示什么并不明显，或者它们为什么省略并不清楚。

也许，在理想情况下，这个 RemoteCall()方法有多个重载版本，其中一个重载版本仅需要两个参数：

```
RemoteCall(var1, var2);
```

但是，这需要更多带其他参数组合的方法，这本身就会带来更多问题(要维护更多的代码，增加了代码的复杂性等)。

Visual Basic 等语言以另一种方式处理这种情况，即允许使用命名参数和可选参数。在 C# 4 版本中也允许这样做，这是所有 .NET 语言的演化趋于一致的一种方式。

下面几节介绍如何使用这些新的参数类型。

#### 14.5.1 可选参数

调用方法时，常常给某个参数传送相同的值。例如，这可能是一个布尔值，以控制方法操作中的不重要部分。具体而言，考虑下面的方法定义：

```
public List<string> GetWords(
    string sentence,
    bool capitalizeWords)
{
    ...
}
```

无论给 `capitalizeWords` 参数传送什么值，这个方法都会返回一系列 `string` 值，每个 `string` 值都是输入句子中的一个单词。根据这个方法的使用方式，可能需要把返回的单词列表转换为大写(也许要格式化一个标题)。但在大多数情况下，并不需要这么做，所以大多数调用如下所示：

```
List<string> words = GetWords(sentence, false);
```

为了把这种方式变成“默认”方式，可以声明第二个方法，如下所示：

```
public List<string> GetWords(string sentence)
{
    return GetWords(sentence, false);
}
```

这个方法调用第二个方法，并给 `capitalizeWords` 传送值 `false`。

这么做没有任何错误，但可以想象在使用更多的参数时，这种方式会非常复杂。

另一种方式是把 `capitalizeWords` 参数变成可选参数。这需要在方法定义中将参数定义为可选参数，这需要提供一个默认值，如果没有提供值，就使用默认值，如下所示：

```
public List<string> GetWords(
    string sentence,
    bool capitalizeWords = false)
{
    ...
}
```

如果以这种方式定义方法，就可以提供一个或两个参数，只有希望 `capitalizeWords` 是 `true` 时，才需要第二个参数。

### 1. 可选参数的值

如上一节所述，方法定义了一个可选参数，其语法如下所示：

```
<parameterType> <parameterName> = <defaultValue>
```

给 `<defaultValue>` 用作默认值的内容有一些限制：默认值必须是字面值、常量值、新对象实例或者默认值类型值。因此不会编译下面的代码：

```
public bool CapitalizationDefault;

public List<string> GetWords(
    string sentence,
    bool capitalizeWords = CapitalizationDefault)
{
    ...
}
```

为了使上述代码可以工作，`CapitalizationDefault` 值必须定义为常量：

```
public const bool CapitalizationDefault = false;
```

这是否有意义取决于具体的情形，在大多数情况下，最好提供一个字面值，就像上一节那样。

## 2. 可选参数的顺序

使用可选值时，它们必须位于方法的参数列表末尾。没有默认值的参数不能放在有默认值的参数后面。

因此下面的代码是非法的：

```
public List<string> GetWords(
    bool capitalizeWords = false,
    string sentence)
{
    ...
}
```

其中，`sentence` 是必选参数，因此必须放在可选参数 `capitalizedWords` 的前面。

### 14.5.2 命名参数

使用可选参数时，可能会发现某个方法有几个可选参数，但您可能只想给第三个可选参数传送值。从上一节介绍的语法来看，如果不提供前两个可选参数的值，就无法给第三个可选参数传送值。

C# 4 引入了命名参数(named parameters)，它允许指定要使用哪个参数。这不需要在方法定义中进行任何特殊处理，它是一个在调用方法时使用的技术。其语法如下：

```
MyMethod(
    <param1Name>: <param1Value>,
    ...
    <paramNName>: <paramNValue>);
```

参数的名称是在方法定义中使用的变量名。

只要命名参数存在，就可以用这种方式指定需要的任意多个参数，而且参数的顺序是任意的。命名参数也是可选的。

可以仅给方法调用中的某些参数使用命名参数。当方法签名中有多个可选参数和一些必选参数时，这是非常有用的。可以先指定必选参数，再指定命名的可选参数。

例如：

```
MyMethod(
    requiredParameter1Value,
    optionalParameter5: optionalParameter5Value);
```

但注意，如果混合使用命名参数和位置参数，就必须先包含所有的位置参数，其后是命名参数。只要使用命名参数，参数的顺序也可以不同。例如：

```
MyMethod(
    optionalParameter5: optionalParameter5Value,
    requiredParameter1: requiredParameter1Value);
```

此时，必须包含所有必选参数的值。

下面的示例介绍了如何使用命名参数和可选参数。

**试一试：使用命名参数和可选参数**

(1) 创建一个新的控制台应用程序 `Ch14Ex04`，把它保存在 `C:\BegVCSharp\Chapter14` 目录中。

(2) 在项目中添加一个类 WordProcessor, 修改其代码, 如下所示:



可从  
wrox.com  
下载源代码

```
public static class WordProcessor
{
    public static List<string> GetWords(
        string sentence,
        bool capitalizeWords = false,
        bool reverseOrder = false,
        bool reverseWords = false)
    {
        List<string> words = new List<string>(sentence.Split(' '));
        if (capitalizeWords)
            words = CapitalizeWords(words);
        if (reverseOrder)
            words = ReverseOrder(words);
        if (reverseWords)
            words = ReverseWords(words);
        return words;
    }

    private static List<string> CapitalizeWords(
        List<string> words)
    {
        List<string> capitalizedWords = new List<string>();
        foreach (string word in words)
        {
            if (word.Length == 0)
                continue;
            if (word.Length == 1)
                capitalizedWords.Add(
                    word[0].ToString().ToUpper());
            else
                capitalizedWords.Add(
                    word[0].ToString().ToUpper()
                    + word.Substring(1));
        }

        return capitalizedWords;
    }

    private static List<string> ReverseOrder(List<string> words)
    {
        List<string> reversedWords = new List<string>();
        for (int wordIndex = words.Count - 1;
            wordIndex >= 0; wordIndex--)
            reversedWords.Add(words[wordIndex]);

        return reversedWords;
    }

    private static List<string> ReverseWords(List<string> words)
    {
        List<string> reversedWords = new List<string>();
        foreach (string word in words)
            reversedWords.Add(ReverseWord(word));
    }
}
```

```

        return reversedWords;
    }

    private static string ReverseWord(string word)
    {
        StringBuilder sb = new StringBuilder();
        for (int characterIndex = word.Length - 1;
            characterIndex >= 0; characterIndex--)
            sb.Append(word[characterIndex]);

        return sb.ToString();
    }
}

```

代码段 Ch14Ex04\WordProcessor.cs

(3) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

static void Main(string[] args)
{
    string sentence = "'twas brillig, and the slithy toves did gyre "
        + "and gimble in the wabe:";
    List<string> words;

    words = WordProcessor.GetWords(sentence);
    Console.WriteLine("Original sentence:");
    foreach (string word in words)
    {
        Console.Write(word);
        Console.Write(' ');
    }

    Console.WriteLine('\n');

    words = WordProcessor.GetWords(
        sentence,
        reverseWords: true,
        capitalizeWords: true);
    Console.WriteLine("Capitalized sentence with reversed words:");
    foreach (string word in words)
    {
        Console.Write(word);
        Console.Write(' ');
    }

    Console.ReadKey();
}

```

代码段 Ch14Ex04\Program.cs

(4) 运行应用程序，结果如图 14-9 所示。

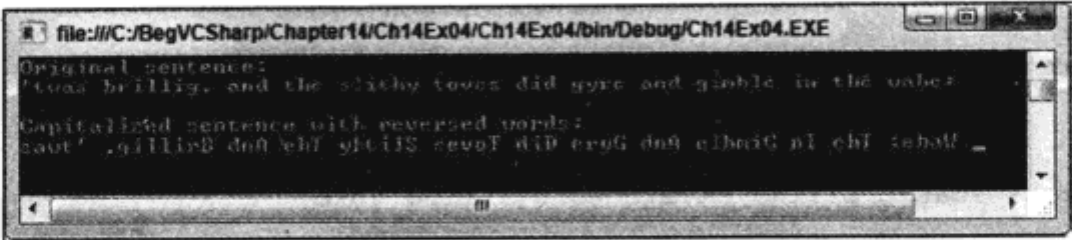


图 14-9

示例的说明

这个示例创建了一个实用类，它执行一些简单的字符串处理，再使用这个类修改一个字符串。类中的单个公共方法包含一个必选参数和 3 个可选参数：



```
public static List<string> GetWords(  
    string sentence,  
    bool capitalizeWords = false,  
    bool reverseOrder = false,  
    bool reverseWords = false)  
{  
    ...  
}
```

代码段 Ch14Ex04\WordProcessor.cs

这个方法返回一个 string 值的集合，每个 string 值都是初始输入的一个单词。根据指定的 3 个可选参数，可能会进行额外的转换：对字符串集合进行整体转换，或者仅转换某个单词。



这里没有深入探讨 WordProcessor 类的功能，读者可以自己研究它的代码，考虑一下如何改进这些代码，'twas 应改为'Twas 吗？如何进行这个修改？

调用这个方法时，只使用了两个可选参数，第三个参数(reverseOrder)使用其默认值 false：

```
words = WordProcessor.GetWords(  
    sentence,  
    reverseWords: true,  
    capitalizeWords: true);
```

还要注意，所指定的两个参数的顺序与定义它们的顺序不同。最后要注意的是，处理带可选参数的方法时，使用 IntelliSense 会非常方便。输入这个示例的代码时，注意 GetWords()方法的工具提示，如图 14-10 所示(把鼠标指针停放在方法调用上，也会看到这个工具提示)。

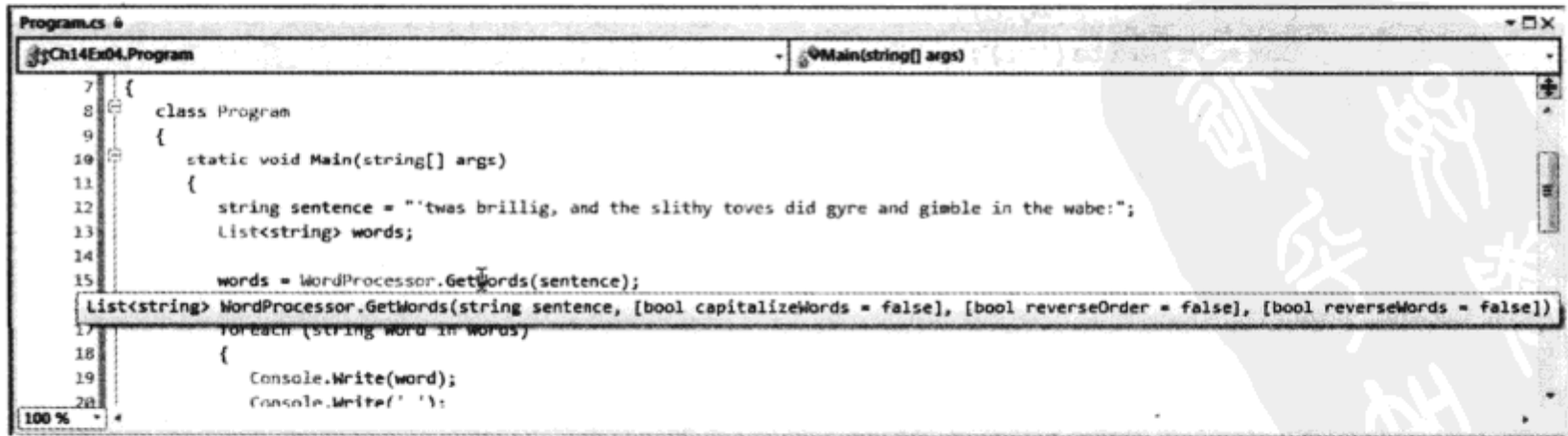


图 14-10

这是一个非常有用的工具提示，它不仅显示了可用参数的名称，还显示了可选参数的默认值，非常便于确定是否需要重写某个默认值。

### 14.5.3 命名参数和可选参数的规则

自从引入了命名参数和可选参数之后，人们对它们的反应不一。一些开发人员，尤其是使用 Microsoft Office 的开发人员，非常喜欢它们；但许多其他的开发人员认为这种修改对 C#语言而言是不必要的，觉得设计良好的用户界面应该不需要这种访问方式——至少在对 C#语言的这层改变上是不需要的。

我个人认为命名参数和可选参数有一些优点，但我担心对它们的过度使用会伤害代码。对于一些情形，例如上面提到的 Microsoft Office，它们肯定是有益的。另外，像上一个示例中的代码那样，定义了许多选项来控制方法的操作，这使代码更容易编写和使用。但在大多数情况下，没有合适的理由，最好不要使用命名参数和可选参数。也许应对方法调用进行充分的测试，看看在事先不知道方法应执行什么操作的情况下，是否能确定会得到什么结果。如果参数及其用法很明显(在编写好的代码中，应很明显)，就不需要使用命名参数或可选参数来修订代码。

## 14.6 扩展方法

扩展方法可以扩展类型的功能，但无需修改类型本身。甚至可以使用扩展方法扩展不能修改的类型，包括在 .NET Framework 中定义的类型。例如，使用扩展方法甚至可以给 `System.String` 等基本类型添加功能。

为了扩展类型的功能，需要提供可以通过该类型的实例调用的方法。为此创建的方法称为扩展方法(extension method)，它可以带任意数量的参数，返回任意类型(包括 `void`)。要创建和使用扩展方法，必须：

- (1) 创建一个非泛型静态类。
- (2) 使用扩展方法的语法，给所创建的类添加扩展方法，作为静态方法(稍后介绍)。
- (3) 确保使用扩展方法的代码用 `using` 语句导入了包含扩展方法类的名称空间。
- (4) 通过扩展类型的一个实例调用扩展方法，与调用扩展类型的其他方法一样。

C#编译器在第(3)步和第(4)步之间完成了它的使命。IDE 会立即发现我们创建了一个扩展方法，并显示在 IntelliSense 中，如图 14-11 所示。

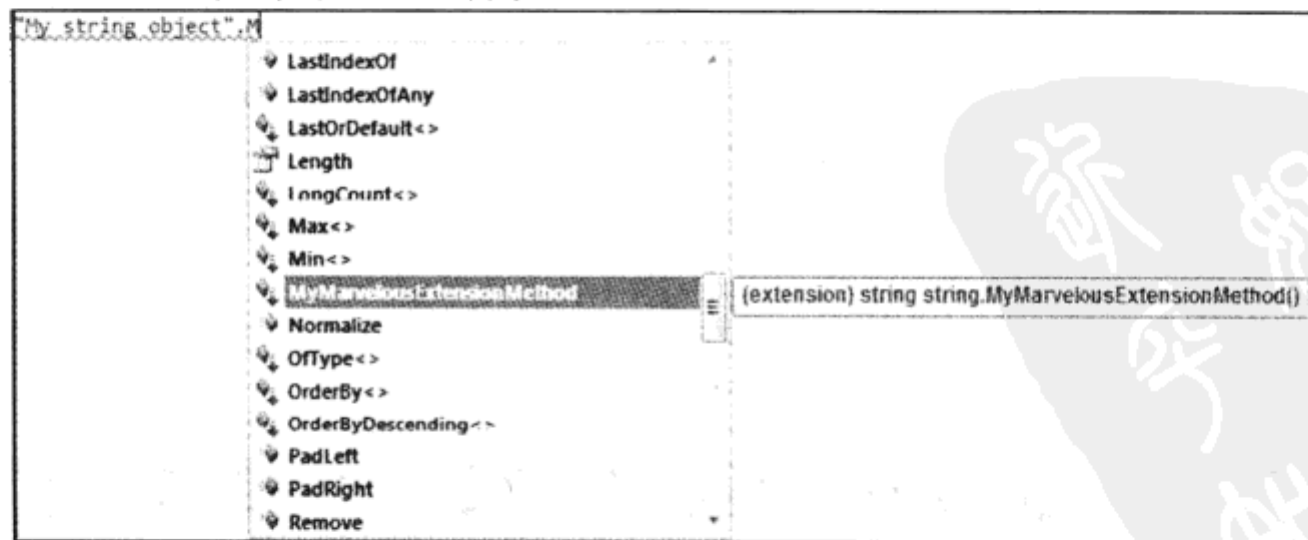


图 14-11



在图 14-11 中,可以通过 `string` 对象(这里仅是一个字面量字符串)使用扩展方法 `MyMarvelousExtensionMethod()`。这个方法用一个略微不同的方法图标来表示,该图标包含一个蓝色的向下箭头,这个方法不带其他参数,返回一个字符串。

为了定义扩展方法,应以与其他方法相同的方式定义一个方法,但该方法必须满足扩展方法的语法要求。这些要求如下:

- 方法必须是静态的。
- 方法必须包含一个参数,表示调用扩展方法的类型实例(这个参数在这里称为实例参数)。
- 实例参数必须是为方法定义的第一个参数。
- 除了 `this` 关键字之外,实例参数不能有其他修饰符。

扩展方法的语法如下:

```
public static class ExtensionClass
{
    public static <ReturnType> <ExtensionMethodName>
        (this <TypeToExtend> instance)
    {
        ...
    }
}
```

导入了包含静态类(其中包括此方法)的名称空间后(也就是使扩展方法变得可用),就可以编写如下代码:

```
<TypeToExtend> myVar;
// myVar is initialized by code not shown here.
myVar. <ExtensionMethodName> ();
```

还可以在扩展方法中包含需要的其他参数,并使用其返回类型。

这个调用实际上与下面的调用相同,但语法更简单:

```
<TypeToExtend> myVar;
// myVar is initialized by code not shown here.
ExtensionClass. <ExtensionMethodName> (myVar);
```

另一个优点是,导入后,就可以通过 `IntelliSense` 查看匿名类型,这样能更容易地找到需要的功能。扩展方法可能分布在多个扩展类中,甚至分布在多个库中,但它们都会显示在扩展类型的成员列表中。

定义了可以用于某个类型的扩展方法后,还可以把它用于派生于这个类型的子类型。在本章前面的一个示例中,如果为 `Animal` 类定义了一个扩展方法,就可以在诸如 `Cow` 的对象上调用它。

还可以定义在特定接口上执行的扩展方法,接着就可以给实现了该接口的任意类型使用该扩展方法。

扩展方法为在应用程序中重用实用代码库提供了一种方式。它们还可以广泛用于本书后面介绍的 LINQ 中。为了更好地理解它,下面看一个示例。

### 试一试: 定义和使用扩展方法

- (1) 创建一个新的控制台应用程序 `Ch14Ex05`, 把它保存在 `C:\BegVCSharp\Chapter14` 目录中。

(2) 在解决方案中添加一个新的类库项目 ExtensionLib。

(3) 从 ExtensionLib 中删除已有的 Class1.cs 类文件，在项目中添加 Ch14Ex04 中的类文件 WordProcessor.cs。

(4) 修改 WordProcessor.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
namespace ExtensionLib
{
    public static class WordProcessor
    {
        public static List<string> GetWords(
            this string sentence,
            bool capitalizeWords = false,
            bool reverseOrder = false,
            bool reverseWords = false)
        {
            ...
        }
        ...
        public static string ToStringReversed(this object inputObject)
        {
            return ReverseWord(inputObject.ToString());
        }

        public static string AsSentence(this List<string> words)
        {
            StringBuilder sb = new StringBuilder();
            for (int wordIndex = 0; wordIndex < words.Count; wordIndex++)
            {
                sb.Append(words[wordIndex]);
                if (wordIndex != words.Count - 1)
                {
                    sb.Append(' ');
                }
            }
            return sb.ToString();
        }
    }
}
```

代码段 ExtensionLib\WordProcessor.cs

(5) 在 Ch14Ex05 项目中添加对 ExtensionLib 项目的引用。

(6) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using ExtensionLib;

namespace Ch14Ex05
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter a string to convert:");
        }
    }
}
```

```
        string sourceString = Console.ReadLine();
        Console.WriteLine("String with title casing: {0}",
            sourceString.GetWords(capitalizeWords: true)
                .AsSentence());
        Console.WriteLine("String backwards: {0}",
            sourceString.GetWords(reverseOrder: true,
                reverseWords: true).AsSentence());
        Console.WriteLine("String length backwards: {0}",
            sourceString.Length.ToStringReversed());
        Console.ReadKey();
    }
}
```

代码段 Ch14Ex05\Program.cs

(7) 运行应用程序。当出现提示时，键入一个字符串(至少 10 个字符长，最好包含多个单词)。结果如图 14-12 所示。

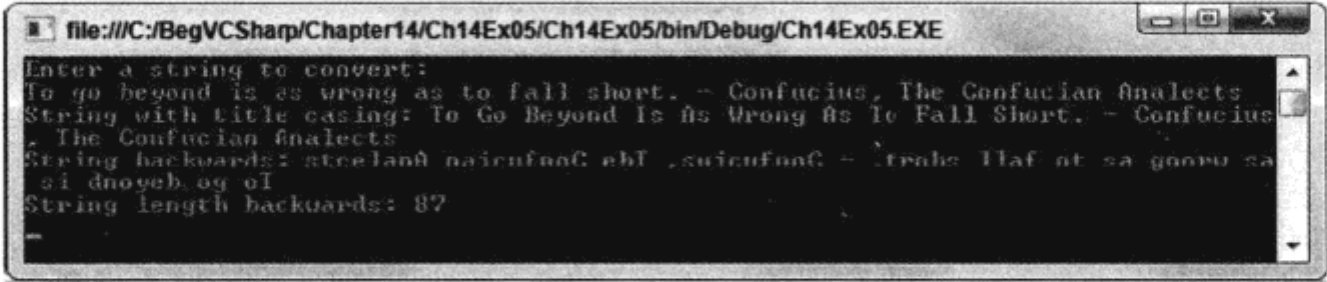


图 14-12

示例的说明

这个示例创建了一个类库，其中包含在一个简单的客户应用程序中使用的实用扩展方法。这个类库有一个静态类 WordProcessor 的扩展版本，WordProcessor 来自上一个包含扩展方法的示例，我们把包含这个类的名称空间 ExtensionLib 导入客户应用程序，以便使用这些扩展方法。

我们创建了 3 个扩展方法，如表 14-1 中所示。

表 14-1

方 法	用 法
GetWords()	操作字符串的灵活方法，如上一个示例所示。在这个示例中，这个方法改为扩展方法，返回一个 List<string>
ToStringReversed()	对于在对象上调用 ToString()所返回的字符串，使用 ReverseWord()使其中的字母逆序，并返回一个字符串
AsSentence()	“铺平”一个 List<string>对象，返回一个字符串，该字符串由它包含的单词组成

客户代码依次调用这些方法，以各种方式修改输入的字符串。前面定义的 GetWords()方法返回一个 List<string>，所以使用 AsSentence()把它的输出结果铺平为一个字符串，以便于使用。

扩展方法 ToStringReversed()是一个比较一般的扩展方法，它不需要 string 类型的实例参数，而是有一个 object 类型的实例参数。这表示这个扩展方法可以在任意对象上调用，显示在所使用的每个对象的 IntelliSense 中。在这个扩展方法中没有做太多的工作，因为不能对要使用的对象做太多的假定。可以使用 is 运算符或尝试类型转换，来确定实例参数的类型，并据此执行相应的操作，也可

以执行这个例子的操作，使用所有对象都支持的基本功能——ToString()方法：

```
public static string ToStringReversed(this object inputObject)
{
    return ReverseWord(inputObject.ToString());
}
```

这个方法只是在其实例参数上调用了 ToString()方法，用前面的 ReverseWord()方法使实例参数中的字母逆序。在示例客户程序中，从 int 变量上调用了 ToStringReversed()方法，得到该整数的一个字符串表示，其中的数字顺序被颠倒了。

可以在多个类型上使用的扩展方法非常有用。另外，还可以定义泛型扩展方法，它们可以把约束应用于类型，如第 12 章所述。

## 14.7 Lambda 表达式

Lambda 表达式是 C# 3.0 引入的一个结构，可用于简化 C#编程的某些方面，尤其是与 LINQ 合并的方面。Lambda 表达式一开始很难掌握，主要是因为其用法非常灵活。Lambda 表达式与其他 C#语言特性(如匿名方法)结合使用时尤其有用。由于本书后面才介绍 LINQ，因此匿名方法是介绍 Lambda 表达式的最佳切入点。下面首先概述一下匿名方法。

### 14.7.1 复习匿名方法

第 13 章学习了匿名方法，这是提供的内联(inline)方法，否则就需要使用委托类型的变量。给事件添加处理程序时，过程如下：

- (1) 定义一个事件处理方法，其返回类型和参数匹配要订阅的事件需要的委托的返回类型和参数。
- (2) 声明一个委托类型的变量，用于事件。
- (3) 把委托变量初始化为委托类型的实例，该实例指向事件处理方法。
- (4) 把委托变量添加到事件的订阅者列表中。

实际上，这个过程会比上述简单一些，因为一般不使用变量存储委托，只在订阅事件时使用委托的一个实例。

下面是第 13 章使用的代码：

```
Timer myTimer = new Timer(100);
myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

这段代码订阅了 Timer 对象的 Elapsed 事件。这个事件使用委托类型 ElapsedEventHandler，使用方法标识符 WriteChar 实例化该委托类型。结果是 Timer 对象引发 Elapsed 事件时，就调用方法 WriteChar()。传送给 WriteChar()的参数取决于由 ElapsedEventHandler 委托定义的类型和 Timer 中引发事件的代码传送的值。

实际上，如第 13 章所述，C#编译器可以通过方法组语法，用更少的代码获得相同的结果：

```
myTimer.Elapsed += WriteChar;
```

C#编译器知道 Elapsed 事件需要的委托类型，所以可以填充该类型。但是，在大多数情况下，最好不要这么做，因为这会使代码更难理解，也不清楚会发生什么。使用匿名方法时，该过程会减少为一步：

(1) 使用内联的匿名方法，该匿名方法的返回类型和参数匹配所订阅事件需要的委托的返回类型和参数。

用 delegate 关键字定义内联的匿名方法：

```
myTimer.Elapsed +=
    delegate(object source, ElapsedEventArgs e)
    {
        Console.WriteLine(
            "Event handler called after {0} milliseconds.",
            (source as Timer).Interval);
    };
```

这段代码可以正常工作，且使用了事件处理程序。主要区别是这里使用的匿名方法对于其余代码而言实际上是隐藏的。例如，不能在应用程序的其他地方重用这个事件处理程序。另外，为了更好地加以描述，这里使用的语法有点沉闷。delegate 关键字总是会带来混淆，因为它有双重含义——匿名方法和定义委托类型都要使用它。

#### 14.7.2 把 Lambda 表达式用于匿名方法

下面看看 Lambda 表达式。Lambda 表达式是简化匿名方法的语法的一种方式。实际上，Lambda 表达式还有其他用处，但为了简单起见，本节只介绍 Lambda 表达式的这个方面。使用 Lambda 表达式可以重写上一节最后的一段代码，如下所示：

```
myTimer.Elapsed += (source, e) => Console.WriteLine(
    "Event handler called after {0} milliseconds.",
    (source as Timer).Interval);
```

这段代码初看上去还好，只是有点让人摸不着头脑(除非很熟悉所谓的函数化编程语言，如 Lisp 或 Haskell)。但是，如果仔细观察，就会看出或至少推断出代码是如何工作的，它与所替代的匿名方法有什么关系。Lambda 表达式由 3 个部分组成：

- 放在括号中的参数列表(未类型化)
- => 运算符
- C# 语句

使用本章前面“匿名类型”一节中介绍的逻辑，从上下文中推断出参数的类型。=> 运算符只是把参数列表与表达式体分开。在调用 Lambda 表达式时，执行表达式体。

编译器会提取这个 Lambda 表达式，创建一个匿名方法，其工作方式与上一节中的匿名方法相同。其实，它会被编译为相同或相似的 CIL 代码。

为了说明 Lambda 表达式中的内容，下面举一个例子。

**试一试：使用简单的 Lambda 表达式**

- (1) 创建一个新的控制台应用程序 Ch14Ex06，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 修改 Program.cs 中的代码, 如下所示:



```
namespace Ch14Ex04
{
    delegate int TwoIntegerOperationDelegate(int paramA, int paramB);

    class Program
    {
        static void PerformOperations(TwoIntegerOperationDelegate del)
        {
            for (int paramAVal = 1; paramAVal <= 5; paramAVal++)
            {
                for (int paramBVal = 1; paramBVal <= 5; paramBVal++)
                {
                    int delegateCallResult = del(paramAVal, paramBVal);
                    Console.WriteLine("f({0},{1})={2}",
                        paramAVal, paramBVal, delegateCallResult);
                    if (paramBVal != 5)
                    {
                        Console.WriteLine(",");
                    }
                }
                Console.WriteLine();
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("f(a, b) = a + b: ");
            PerformOperations((paramA, paramB) => paramA + paramB);
            Console.WriteLine();
            Console.WriteLine("f(a, b) = a * b: ");
            PerformOperations((paramA, paramB) => paramA * paramB);
            Console.WriteLine();
            Console.WriteLine("f(a, b) = (a - b) % b: ");
            PerformOperations((paramA, paramB) => (paramA - paramB)
                % paramB);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch14Ex06\Program.cs

(3) 运行应用程序, 结果如图 14-13 所示。



图 14-13

示例的说明

这个示例使用 Lambda 表达式生成函数，在两个输入参数上执行指定的处理，并返回结果。接着这些函数操作 25 对值，把结果输出到控制台上。

首先定义一个委托类型 TwoIntegerOperationDelegate，表示一个方法，该方法有两个 int 参数，返回一个 int 结果：

```
delegate int TwoIntegerOperationDelegate(int paramA, int paramB);
```

在以后定义 Lambda 表达式时使用这个委托类型。这些 Lambda 表达式编译为方法，其返回类型和参数匹配这个委托类型，如稍后所述。

接着添加方法 PerformOperations()，它带有一个 TwoIntegerOperationDelegate 类型的参数：

```
static void PerformOperations(TwoIntegerOperationDelegate del)
{
```

这个方法的含义是，可以给它传送一个委托实例(或者匿名方法，或者 Lambda 表达式，因为这些结构都会编译为委托实例)，该方法会用一组值调用委托实例所表示的方法：

```
for (int paramAVal = 1; paramAVal <= 5; paramAVal++)
{
    for (int paramBVal = 1; paramBVal <= 5; paramBVal++)
    {
        int delegateCallResult = del(paramAVal, paramBVal);
```

接着把参数和结果输出到控制台上：

```
        Console.Write("f({0},{1})={2}",
            paramAVal, paramBVal, delegateCallResult);
        if (paramBVal != 5)
        {
            Console.Write(",");
        }
    }
    Console.WriteLine();
}
```



在 Main()方法中, 创建了 3 个 Lambda 表达式, 使用它们依次调用 PerformOperations(), 第一个调用如下所示:

```
Console.WriteLine("f(a, b) = a + b: ");
PerformOperations((paramA, paramB) => paramA + paramB);
```

这里使用的 Lambda 表达式如下:

```
(paramA, paramB) => paramA + paramB
```

这个 Lambda 表达式分为 3 部分:

(1) 参数定义部分。这里有两个参数 paramA 和 paramB。这些参数都是未类型化的, 因此编译器可以根据上下文推断出它们的类型。在这个例子中, 编译器可以确定, PerformOperations()方法调用需要一个 TwoIntegerOperationDelegate 类型的委托。这个委托类型有两个 int 参数, 所以根据推断, paramA 和 paramB 都是 int 类型的变量。

(2) =>运算符。它把 Lambda 表达式的参数与表达式体分开。

(3) 表达式体。它指定了一个简单的操作: 把 paramA 和 paramB 加起来。注意, 不需要指定这是返回值。编译器知道要创建可以使用 TwoIntegerOperationDelegate 的方法, 这个方法就必须有 int 返回类型。根据指定的操作, paramA + paramB 等于一个 int 类型的值, 且没有提供额外的信息, 所以编译器推断, 这个表达式的结果就是方法的返回类型。

接着, 就可以把使用这个 Lambda 表达式的代码扩展到下面使用匿名方法的代码中:

```
Console.WriteLine("f(a, b) = a + b:");
PerformOperations(delegate(int paramA, int paramB)
{
    return paramA + paramB;
});
```

其余代码以相同的方式使用两个不同的 Lambda 表达式来执行操作:

```
Console.WriteLine();
Console.WriteLine("f(a, b) = a * b: ");
PerformOperations((paramA, paramB) => paramA * paramB);
Console.WriteLine();
Console.WriteLine("f(a, b) = (a - b) % b: ");
PerformOperations((paramA, paramB) => (paramA - paramB)
    % paramB);
Console.ReadKey();
```

最后一个 Lambda 表达式涉及较多的计算, 但并不比其他 Lambda 表达式更复杂。Lambda 表达式的语法允许执行更复杂的操作, 如稍后所述。

### 14.7.3 Lambda 表达式的参数

在前面的代码中, Lambda 表达式使用类型推理功能确定所传送的参数类型。实际上这不是必须的, 也可以定义类型。例如, 可以使用下面的 Lambda 表达式:

```
(int paramA, int paramB) => paramA + paramB
```

其优点是代码更容易理解, 但不够简明灵活。在前面委托类型的示例中, 可以通过隐式类型化

的 Lambda 表达式来使用其他数字类型，例如，long 变量。

注意，不能在同一个 Lambda 表达式中同时使用隐式和显式的参数类型。下面的 Lambda 表达式就不会编译，因为 paramA 是显式类型化的，而 paramB 是隐式类型化的：

```
(int paramA, paramB) = > paramA + paramB
```

Lambda 表达式的参数列表总是包含一个用逗号分隔的列表，其中的参数要么都是显式类型化的，要么都是隐式类型化的。如果只有一个参数，就可以省略括号；否则就需要在参数列表上加上括号，如前面所示。例如，下面的 Lambda 表达式只有一个参数，且是隐式类型化的：

```
param1 = > param1 * param1
```

还可以定义没有参数的 Lambda 表达式，这使用空括号来表示：

```
() = > Math.PI
```

当委托不需要参数，但需要返回一个 double 值时，就可以使用这个 Lambda 表达式。

#### 14.7.4 Lambda 表达式的语句体

在前面的所有代码中，Lambda 表达式的语句体都只使用了一个表达式。并说明了这个表达式如何解释为 Lambda 表达式的返回值，例如，如何给返回类型为 int 的委托使用表达式 paramA+paramB 作为 Lambda 表达式的语句体(假定 paramA 和 paramB 隐式或显式类型化为 int 值，如示例代码所示)。

前面的一个示例说明了对于语句体中使用的代码而言，返回类型为 void 的委托的要求并不高：

```
myTimer.Elapsed += (source, e) = > Console.WriteLine(
    "Event handler called after {0} milliseconds.",
    (source as Timer).Interval);
```

上面的语句不返回任何值，所以它只是执行，其返回值不在任何地方使用。

Lambda 表达式可以看作匿名方法语法的扩展，所以还可以在 Lambda 表达式的语句体中包含多个语句。为此，只需把一个代码块放在花括号中，类似于 C# 中提供多行代码的其他情况：

```
(param1, param2) = >
{
    // Multiple statements ahoy!
}
```

如果使用 Lambda 表达式和返回类型不是 void 的委托类型，就必须用 return 关键字返回一个值，这与其他方法一样：

```
(param1, param2) = >
{
    // Multiple statements ahoy!
    return returnValue ;
}
```

例如，可以把前面示例中的如下代码：

```
PerformOperations((paramA, paramB) = > paramA + paramB);
```

改写为:

```
PerformOperations(delegate(int paramA, int paramB)
{
    return paramA + paramB;
});
```

另外,也可以把代码改写为:

```
PerformOperations((paramA, paramB) =>
{
    return paramA + paramB;
});
```

这更像是原来的代码,因为它包含 paramA 和 paramB 参数的隐式类型化。

在大多数情况下,使用单一的表达式时,大都使用 Lambda 表达式,它们肯定是最简洁的。说实话,如果需要多个语句,则定义一个非匿名方法来替代 Lambda 表达式比较好,这也会使代码更便于重用。

#### 14.7.5 Lambda 表达式用作委托和表达式树

前面提到了 Lambda 表达式和匿名方法的一些区别:匿名方法比较灵活,例如,隐式类型化的参数。目前,应注意另一个重要区别,但在学习本书后面的 LINQ 之前,这个区别并不是很明显。

可以用两种方式来解释 Lambda 表达式。第一,如本章所述,Lambda 表达式是一个委托。即可以把 Lambda 表达式赋予一个委托类型的变量,如前面的示例所示。

一般可以把拥有至多 8 个参数的 Lambda 表达式表示为如下泛型类型,它们都在 System 命名空间中定义:

- Action 表示的 Lambda 表达式不带参数,返回类型是 void
- Action<>表示的 Lambda 表达式有至多 8 个参数,返回类型是 void
- Func<>表示的 Lambda 表达式有至多 8 个参数,返回类型不是 void

Action<>有至多 8 个泛型类型的参数,分别用于 Lambda 表达式的 8 个参数,Func<>有至多 9 个泛型类型的参数,分别用于 Lambda 表达式的 8 个参数和返回类型。在 Func<>中,返回类型总是在列表的最后。

例如,下面的 Lambda 表达式:

```
(int paramA, int paramB) => paramA + paramB
```

可以表示为 Func<int,int,int>类型的委托,因为它有两个 int 参数,返回类型是 int。

第二,可以把 Lambda 表达式解释为表达式树。表达式树是 Lambda 表达式的抽象表示,但不能直接执行。可以使用表达式树以编程方式分析 Lambda 表达式,执行操作,以响应 Lambda 表达式。

显然这是一个复杂的主题,但表达式树对本书后面介绍的 LINQ 功能至关重要。下面给出一个具体的例子。LINQ 架构包含一个泛型类 Expression<>,可用于封装 Lambda 表达式。使用这个类的一种方式是提取用 C#编写的 Lambda 表达式,把它转换为相应的 SQL 脚本,以便在数据库中直接执行。

目前并不需要了解太多的内容,在本书后面遇到这个功能时,能更好地理解其过程,因为现在

我们已经理解了 C# 提供的重要概念。

### 14.7.6 Lambda 表达式和集合

学习了 `Func<T>` 泛型委托之后，就可以理解 `System.Linq` 名称空间为数组类型提供一些扩展方法了(在编码的不同地方，可以在弹出 `IntelliSense` 时看到它们)。例如，有一个扩展方法 `Aggregate()` 定义了 3 个重载版本，如下所示：

```
public static TSource Aggregate < TSource > (
    this IEnumerable < TSource > source,
    Func < TSource, TSource, TSource > func);

public static TAccumulate Aggregate < TSource, TAccumulate > (
    this IEnumerable < TSource > source,
    TAccumulate seed,
    Func < TAccumulate, TSource, TAccumulate > func);

public static TResult Aggregate < TSource, TAccumulate,
                                   TResult > (
    this IEnumerable < TSource > source,
    TAccumulate seed,
    Func < TAccumulate, TSource, TAccumulate > func,
    Func < TAccumulate, TResult > resultSelector);
```

与前面的扩展方法一样，这段代码初看上去非常深奥，但如果分解它们，就很容易理解其工作过程。这个函数的 `IntelliSense` 告诉用户它会执行如下工作：

Applies an accumulator function over a sequence.

这表示要把一个累加器函数(可以以 `Lambda` 表达式的形式提供)应用于集合中从开始到结束的每对元素上，并把计算的结果作为下一个计算操作的输入。

在 3 个重载版本中，最简单的版本只有一个泛型类型，这可以从实例参数的类型推理出来。例如，在下面的代码中，泛型类型是 `int`(累加器函数现在是空白的)：

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate(...);
```

这等价于：

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate <int> (...);
```

这里需要的 `Lambda` 表达式可以从扩展方法中推断出来。在这段代码中，类型 `TSource` 是 `int`，所以必须为委托 `Func<int,int,int>` 提供一个 `Lambda` 表达式。例如，可以使用前面的 `Lambda` 表达式：

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate((paramA, paramB) => paramA + paramB);
```

这个调用会使 `Lambda` 表达式调用两次，一次使用的参数是 `paramA=2`, `paramB=6`，另一次使用的参数是 `paramA=8`(第一次计算的结果), `paramB=3`。最后赋予变量 `result` 的结果是 `int` 值 11，即数组中所有元素的总和。

扩展方法 `Aggregate()` 的其他两个重载版本是类似的，但可以执行略微复杂的计算，如下面的简短示例所示。

#### 试一试：使用 Lambda 表达式和集合

- (1) 创建一个新的控制台应用程序 `Ch14Ex07`，把它保存在 `C:\BegVCSharp\Chapter14` 目录中。
- (2) 修改 `Program.cs` 中的代码，如下所示：



可从  
WROX.COM  
下载源代码

```
static void Main(string[] args)
{
    string[] curries = { "pathia", "jalfreze", "korma" };
    Console.WriteLine(curries.Aggregate(
        (a, b) => a + " " + b));
    Console.WriteLine(curries.Aggregate < string, int > (
        0,
        (a, b) => a + b.Length));
    Console.WriteLine(curries.Aggregate < string, string, string > (
        "Some curries: ",
        (a, b) => a + " " + b,
        a => a));
    Console.WriteLine(curries.Aggregate < string, string, int > (
        "Some curries: ",
        (a, b) => a + " " + b,
        a => a.Length));
    Console.ReadKey();
}
```

代码段 Ch14Ex07\Program.cs

- (3) 运行应用程序，结果如图 14-14 所示。

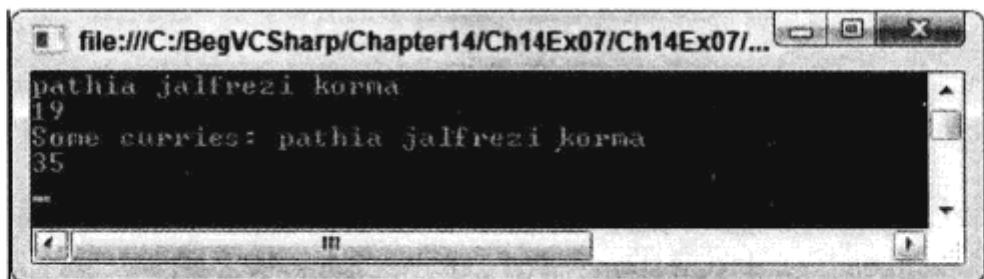


图 14-14

#### 示例的说明

这个示例把包含 3 个元素的字符串数组作为源数据，试验了扩展方法 `Aggregate()` 的每个重载版本。

首先执行一个简单的连接操作：

```
Console.WriteLine(curries.Aggregate(
    (a, b) => a + " " + b));
```

第一对元素用简单的语法串联成一个字符串。这不是连接字符串的最佳方式，最好使用 `string.Concat()` 或 `string.Format()` 优化性能，但这里使用它提供了一种非常简单的方式，来说明发生了什么。第一个串联操作之后，结果传送回 `Lambda` 表达式和数组中的第 3 个元素，其方式与前面要

计算总和的 `int` 值相同。结果是串联整个数组，并用空格分隔各个项。

接着，使用 `Aggregate()` 函数的第二个重载版本，它有两个泛型类型的参数 `TSource` 和 `TAccumulate`。在这个示例中，`Lambda` 表达式的形式必须是 `Func<TAccumulate, TSource, TAccumulate>`。另外，必须指定 `TAccumulate` 类型的种子值，这个种子值在 `Lambda` 表达式的第一个调用和第一个数组元素中使用。后续的调用从前面的调用中把累加器的结果提取到表达式中。代码如下：

```
Console.WriteLine(curries.Aggregate < string, int > (
    0,
    (a, b) => a + b.Length));
```

累加器(以及返回值)的类型是 `int`。累加器的值最初设置为种子值 0，在对 `Lambda` 表达式的每次调用中，都把该值累加到数组元素的长度上。最后的结果是数组中每个元素的总长度。

之后使用 `Aggregate()` 函数的最后一个重载版本，它带有 3 个泛型类型的参数，与其他重载版本的唯一区别是，其返回类型可以与数组元素和累加值的类型都不同。首先，这个重载版本把字符串元素与种子字符串串联在一起：

```
Console.WriteLine(curries.Aggregate < string, string, string > (
    "Some curries:",
    (a, b) => a + " " + b,
    a => a));
```

即使累加值只是复制到结果中，也必须指定这个方法的最后一个参数 `resultSelector`(如本例所示)。这个参数是一个 `Func<TAccumulate, TResult>` 类型的 `Lambda` 表达式。

在最后一段代码中，再次使用了 `Aggregate()` 的这个版本，但这次使用 `int` 类型的返回值。其中，给 `resultSelector` 提供一个 `Lambda` 表达式，返回累加字符串的长度：

```
Console.WriteLine(curries.Aggregate < string, string, int > (
    "Some curries:",
    (a, b) => a + " " + b,
    a => a.Length));
```

这个示例没有什么有趣的地方，但演示了如何使用更复杂的扩展方法，其中涉及泛型类型的参数、集合和看似复杂的语法。本书后面还要讨论它们。

## 14.8 小结

本章介绍了 VS2010 和 Visual C# Express 2010 中使用的 C# 4 语言的新特性，这些特性简化了实现常用或高级功能所需的某些编码。

本章的主要内容如下：

- 如何使用对象和集合初始化器，在一个步骤中完成对象和集合的实例化和初始化。
- IDE 和 C# 编译器如何从上下文中推断类型，如何使用 `var` 关键字把类型推理功能用于任意变量类型。
- 如何创建和使用匿名类型，它们合并了前面介绍的初始化器和类型推理主题。
- 如何在变量上使用动态查找功能，该变量仅在运行期间访问，以用于成员。



- 如何使用命名参数和可选参数，以灵活的方式调用方法。
- 如何创建扩展方法(扩展方法可以在其他类型的实例上调用，且无需在这些类型的定义中添加代码)，如何使用这个技术提供实用方法库。
- 如何使用 Lambda 表达式给委托实例提供匿名方法，Lambda 方法的扩展语法如何实现其他功能。

本章介绍的大多数 C# 特性都已添加，以满足 .NET Framework 的 LINQ 新功能的需要。本章中代码的许多子主题以后会进一步阐明。如前所述，我们学习了一些非常强大的技术，可以用于快速提升 C# 编程技巧。

现在，我们已经完成了 C# 4 语言的全部介绍，但这并不意味着，这些内容囊括了 .NET Framework 编程的所有内容。C# 语言提供了编写 .NET 应用程序所需的所有工具，但 .NET Framework 中的类为您提供建立了应用程序的原材料。本书从现在开始，就要深入探讨这些类，学习如何使用它们执行各种任务了。第 15 章不再使用前面大量使用的控制台应用程序，而开始使用 Windows Forms 提供的丰富功能创建图形化的用户界面。此时应注意尽管所创建的应用程序类型不同，但底层的规则是相同的，本书第 I 部分介绍的技巧也可以在后面的章节中使用。

## 14.9 练习

(1) 为什么不能把对象初始化器用于下面的类？修改这个类，使之能使用对象初始化器。之后给出一个示例代码，仅通过一个步骤实例化和初始化这个类：

```
public class Giraffe
{
    public Giraffe(double neckLength, string name)
    {
        NeckLength = neckLength;
        Name = name;
    }
    public double NeckLength {get; set;}
    public string Name {get; set;}
}
```

- (2) 判断正误：如果声明一个 var 类型的变量，就可以使用它存储任意对象类型。
- (3) 使用匿名类型时，如何比较两个实例，确定它们是否包含相同的数据？
- (4) 更正下述扩展方法的代码，其中包含一个错误：

```
public string ToAcronym(this string inputString)
{
    inputString = inputString.Trim();
    if (inputString == "")
    {
        return "";
    }
    string[] inputStringAsArray = inputString.Split(' ');
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < inputStringAsArray.Length; i++)
    {
        if (inputStringAsArray[i].Length > 0)
        {
            sb.AppendFormat("{0}", inputStringAsArray[i].Substring(0, 1).ToUpper());
        }
    }
}
```



```
    }  
    return sb.ToString();  
}
```

(5) 如何确保题(4)中的扩展方法可用于客户代码？

(6) 把题(4)中的 ToAcronym 方法改为一行代码。该代码应确保单词之间包含多个空格的字符串不出错。提示：需要使用?:三元运算符、string.Aggregate<string, string>扩展方法和一个 Lambda 表达式。

附录 A 给出了练习答案。

14.10 本章要点

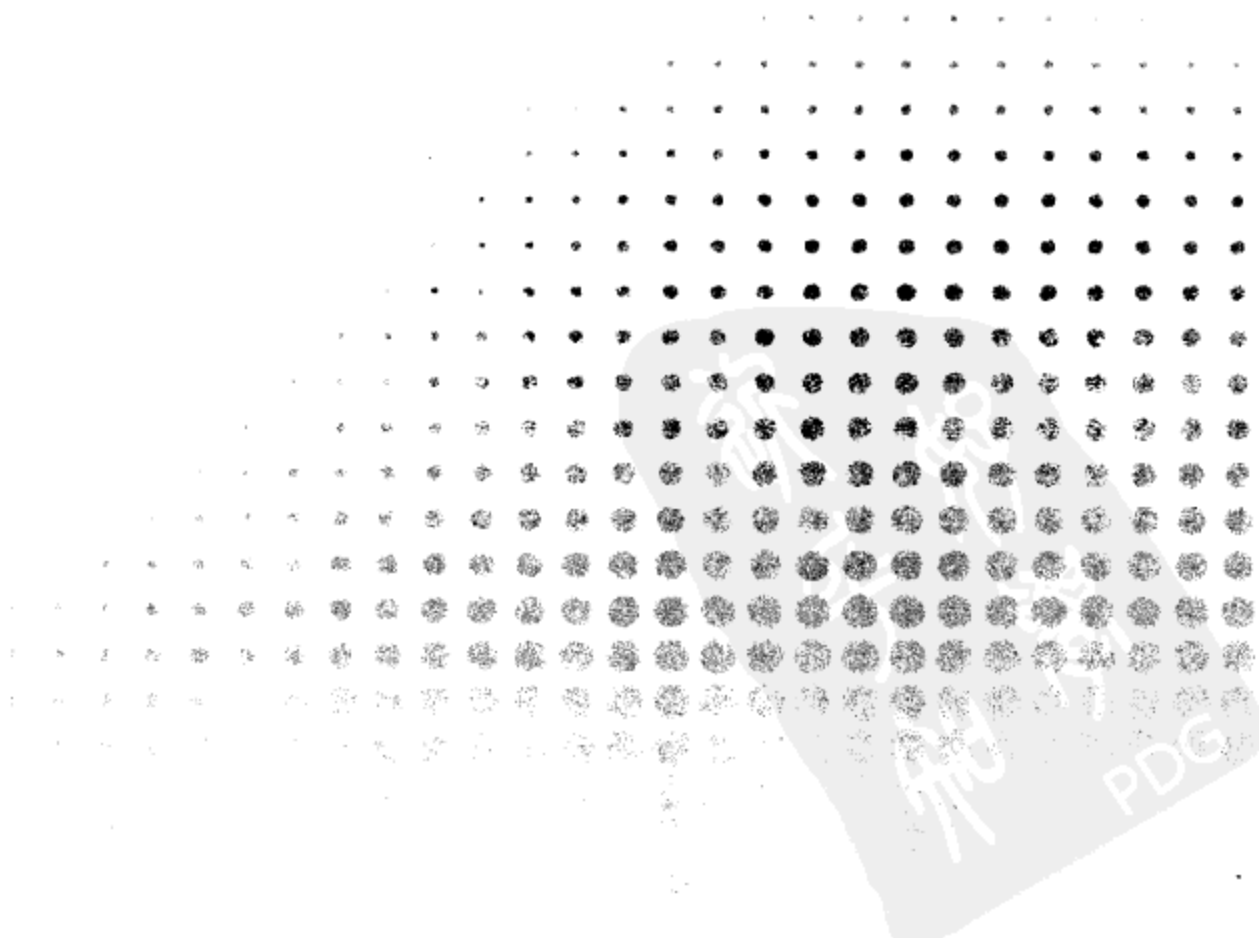
主 题	重 要 概 念
初始化器	可以使用初始化器在创建对象或集合的同时初始化它们。这两种初始化器都包括一个放在花括号中的代码块。对象初始化器可以提供一个逗号分隔的属性名/值对列表，来设置属性值。集合初始化器仅需要逗号分隔的值列表。使用对象初始化器时，还可以使用非默认的构造函数
类型推理	声明变量时，使用 var 关键字允许忽略变量的类型。但只有类型可以在编译期间确定时才会如此。使用 var 没有违反 C#的强类型化规则，因为用 var 声明的变量只能有一种类型
匿名类型	对于用于结构数据存储的许多简单类型，定义类型是不必要的。而可以使用匿名类型，其成员根据用途来推断。使用对象初始化器语法来定义匿名类型，每个设置的属性都定义为只读属性
动态查找	使用 dynamic 关键字定义 dynamic 类型的变量，可以存储任意值。接着就可以使用一般的属性或方法语法来访问被包含的值的成员，这些成员仅在运行期间检查。如果在运行期间，尝试访问一个不存在的成员，就会抛出一个异常。这种动态的类型化显著简化了访问非.NET 类型或类型信息不能在编译期间获得的.NET 类型的语法。但是，在使用时动态类型要谨慎，因为无法在编译期间检查代码。实现 IDynamicMetaObjectProvider 接口，可以控制动态查找的行为
可选的方法参数	我们常常可以定义带许多参数的方法，但其中的许多参数都很少使用。可以提供多个方法重载，而无不是强制客户代码为很少使用的参数提供值。另外，也可以把这些参数定义为可选参数(并为未指定值的参数提供默认值)。调用方法的客户代码就可以仅指定需要的参数
命名的方法参数	客户代码可以根据位置或名称(或者根据位置和名称，其中位置参数放在前面)来指定方法的参数。命名的参数可以按任意顺序指定。这尤其适用于和可选参数一起使用的场合
扩展方法	可以为任意已有的类型定义扩展方法，而无需修改类型定义。例如，这包括扩展系统定义的类型，如 string。扩展方法定义为非泛型静态类的静态方法。实例方法的第一个参数使用 this 关键字定义，是针对其调用方法的实例值。定义完后，扩展方法就可以在任意代码中调用，但这些代码必须引用包含定义该方法的类的名称空间。扩展方法可以在方法定义或任意派生类型使用的类型实例上调用，所以可以为类型系列定义通用的扩展方法。创建通用的扩展方法的另一种方式是创建可以通过特定接口使用的扩展方法
Lambda 表达式	Lambda 表达式实际上是定义匿名方法的一种快捷方式，且有额外的功能，例如隐式的类型化。定义 Lambda 表达式时，需要使用逗号分隔的参数列表(如果没有参数，就使用空括号)、=>运算符和一个表达式。该表达式可以是放在花括号中的代码块。Lambda 表达式至多可以有 8 个参数和一个可选的返回类型，Lambda 表达式可以用 Action、Action<>和 Func<>委托类型来表示。许多可用于集合的 LINQ 扩展方法都使用 Lambda 表达式参数

## 第 II 部分

# Windows 编 程

---

- 第 15 章 Windows 编程基础
- 第 16 章 Windows 窗体的高级功能
- 第 17 章 部署 Windows 应用程序





# 第 15 章

## Windows 编程基础

本章内容:

- Windows 窗体设计器
- 向用户显示信息的控件, 如 Label 和 LinkLabel 控件
- 触发事件的控件, 如 Button 控件
- 允许应用程序的用户输入文本的控件, 如 TextBox 控件
- 允许告诉用户应用程序的当前状态、让用户修改状态的控件, 如 RadioButton 和 CheckButton 控件
- 允许显示信息列表的控件, 如 ListBox 和 ListView 控件
- 允许把其他控件组合在一起的控件, 如 TabControl 和 GroupBox 控件

近 10 年来, Visual Basic 允许程序员使用工具, 通过直观的窗体设计器创建十分复杂的用户界面, 其编程语言的易学易用, 为快速开发应用程序(rapid application development, RAD)提供了尽可能好的环境, 所以赢得了广泛的好评。Visual Basic 等 RAD 工具的一个优点是提供了许多预制控件, 开发人员可以使用它们快速建立应用程序的用户界面。

开发大多数 Visual Basic Windows 应用程序的核心是窗体设计器(Forms Designer)。创建用户界面时, 把控件从工具箱拖放到窗体上, 把它们放在应用程序运行时需要的地方, 再双击该控件, 添加控件的处理程序。Microsoft 提供的控件和可以按合理价格购得的定制控件, 为程序员提供了空前巨大的、已进行了全面测试的重用代码池, 仅通过鼠标单击就可以使用它们。通过 Visual Studio, 这种应用程序开发模式现在也可以用于 C#开发人员。

本章将使用 Windows 窗体, 利用 Visual Studio 附带的许多控件。这些控件拥有各种功能, 通过 Visual Studio 的设计功能, 开发用户界面、处理用户的交互将非常简单、有趣。在本书全面介绍 Visual Studio 中的控件是不可能的, 所以这里只介绍最常用的控件, 包括标签、文本框、列表视图、选项卡控件等。

### 15.1 控件

在使用 Windows 窗体时, 就是在使用 System.Windows.Forms 名称空间。这个名称空间使用 using

指令包含在存储 Form 类的一个文件中。 .NET 中的大多数控件都派生于 System.Windows.Forms.Control 类。这个类定义了控件的基本功能，这就是控件中的许多属性和事件都相同的原因。许多类本身就是其他控件的基类，图 15-1 中的 Label 和 TextBoxBase 类就是这样。

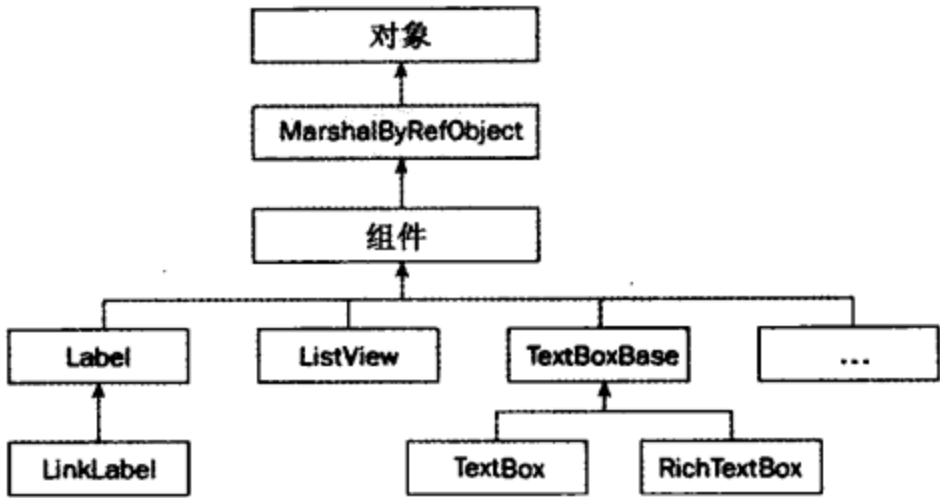


图 15-1

15.1.1 属性

所有控件都有许多属性，用于处理控件的操作。大多数控件的基类都是 System.Windows.Forms.Control，它有许多属性，其他控件要么直接继承了这些属性，要么重写它们以便提供某些定制的操作。

表 15-1 列出了 Control 类最常见的一些属性。这些属性在本章介绍的大多数控件中都有，所以后面将不再详细解释它们，除非属性的操作对于某个控件来说进行了改变。注意这个表并不完整，如果要查看该类的所有属性，请参阅 .NET Framework SDK 文档说明。

表 15-1

属 性	说 明
Anchor	指定当控件的容器大小发生变化时，该控件如何响应。参见下一节来了解对这个属性的详细解释
BackColor	控件的背景色
Bottom	指定控件底部距窗口顶部的距离。这与指定控件的高度不同
Dock	使控件停靠在容器的边界上。参见下面对这个属性的详细解释
Enabled	把 Enabled 设置为 true 通常表示该控件可以接收用户的输入。把 Enabled 设置为 false 通常表示不能接收用户的输入
ForeColor	控件的前景色
Height	控件底部到顶部的距离
Left	控件的左边界距其容器左边界的距离
Name	控件的名称。这个名称可以在代码中用于引用该控件
Parent	控件的父控件
Right	控件的右边界距其容器左边界的距离
TabIndex	控件在容器中的标签顺序号
TabStop	指定是否可以用 Tab 键访问控件

(续表)

属 性	说 明
Tag	这个值通常不由控件本身使用，而是在控件中存储该控件的信息。当通过 Windows 窗体设计器给这个属性赋值时，就只能给它赋一个字符串值
Text	保存与该控件相关的文本
Top	控件顶部距其容器顶部的距离
Visible	指定控件是否在运行期间可见
Width	控件的宽度

15.1.2 控件的定位、停靠和对齐

在 Visual Studio 2005 中，窗体设计器默认情况下不再使用栅格状的界面供设置控件布局，而改用 一个清洁的界面，并且使用捕捉线来定位控件，选择 Tools 菜单上的 Options 选项，就可以在两种设计样式之间切换。在树型视图左侧中选择 Windows Forms Designer 节点，设置 Layout Mode。这完全是一个个人喜好的问题，但在下面的示例中，将使用默认的设计样式。

试一试：使用捕捉线

按照下面的步骤使用 Windows 窗体设计器中的捕捉线：

- (1) 创建一个 Windows 窗体应用程序，命名为 SnapLines。
- (2) 把一个按钮控件从 Toolbox 拖放到窗体的中间位置。
- (3) 把窗体向上拖动到窗体的左上角。注意在接近窗体的边缘时，会从窗体的左边缘和上边缘显示两条线，控件会被固定在该位置上。可以移动控件，使其超过捕捉线的范围，或者就把控件放在这个位置上。
- (4) 把按钮移回到窗体的中心，把另一个按钮从 Toolbox 拖放到窗体上。把它移动到第一个按钮的下面，注意在这个过程中又会出现捕捉线。这些捕捉线可以把控件排列整齐，使控件位于相同的垂直或水平位置上。如果把新按钮向上移近已有的按钮，就会出现另一条捕捉线，允许用预设的距离放置按钮。
- (5) 重新设置 button1 的大小，使它比另一个按钮宽，然后重新设置 button2 的大小，注意当 button2 的宽度与 button1 相同时，就会出现捕捉线，以便把控件的宽度设置为相同的值。
- (6) 现在在按钮的下面给窗体添加一个 TextBox，把其 Text 属性改为 Hello World!。
- (7) 在窗体上添加一个 Label，把它移动到 TextBox 的左边。注意在移动控件时，会出现两条捕捉线，捕捉 TextBox 的顶部和底部，在这两条捕捉线之间，还会出现第三条捕捉线，在把 Label 放在窗体上时，它可以使 TextBox 的文本和 Label 有相同的高度。

15.1.3 Anchor 和 Dock 属性

在设计窗体时，这两个属性特别有用。如果用户认为改变窗口的大小并非小事，应确保窗口看起来不显得很乱；在以前只有编写许多代码行才能达到这个目的。许多程序解决这个问题时，都是禁止给窗口重新设置大小，这显然是解决问题最简单的方法，但不是最好的方法。.NET 引入了 Anchor 和 Dock 属性，就是为了在不编写任何代码的情况下解决这个问题。

Anchor 属性指定在用户重新设置窗口的大小时控件该如何响应。可以指定如果控件重新设置了大小，就根据控件的边界合理地锁定它，或者其大小不变，但根据窗口的边界来锚定它的位置。

Dock 属性指定控件应停靠在容器的边框上。如果用户重新设置了窗口的大小，该控件将继续停放在窗口的边框上。例如，如果指定控件停靠在容器的底部边界上，则无论窗口的大小如何改变，该控件都将改变大小，或移动其位置，确保总是位于屏幕的底部。

本章后面将详细介绍 Anchor 属性。

15.1.4 事件

第 13 章介绍了事件的含义及其用法。本节介绍事件的特定类型，即 Windows 窗体控件生成的控件。这些事件通常与用户的操作相关。例如，在用户单击按钮时，该按钮就会生成一个事件，说明发生了什么。处理事件就是程序员为该按钮提供某些功能的方式。

Control 类定义了本章所用控件的一些比较常见的事件。表 15-2 描述了许多这类事件。这个表仅列出了最常见的事件；如果需要查看完整的列表，请参阅.NET Framework SDK 文档说明。

表 15-2

事 件	说 明
Click	在单击控件时引发。在某些情况下，这个事件也会在用户按下回车键时引发
DoubleClick	在双击控件时引发。处理某些控件上的 Click 事件，如 Button 控件，表示永远不会调用 DoubleClick 事件
DragDrop	在完成拖放操作时引发。换言之，当一个对象被拖到控件上，然后用户释放鼠标按钮后，引发该事件
DragEnter	在被拖动的对象进入控件的边界时引发
DragLeave	在被拖动的对象移出控件的边界时引发
DragOver	在被拖动的对象放在控件上时引发
KeyDown	当控件有焦点时，按下一个键时引发该事件，这个事件总是在 KeyPress 和 KeyUp 之前引发
KeyPress	当控件有焦点时，按下一个键时发生该事件，这个事件总是在 KeyDown 之后、KeyUp 之前引发。KeyDown 和 KeyPress 的区别是 KeyDown 传送被按下的键的键盘码，而 KeyPress 传送被按下的键的 char 值
KeyUp	当控件有焦点时，释放一个键时发生该事件，这个事件总是在 KeyDown 和 KeyPress 之后引发
GotFocus	在控件接收焦点时引发。不要用这个事件执行控件的有效性验证，而应使用 Validating 和 Validated
LostFocus	在控件失去焦点时引发。不要用这个事件执行控件的有效性验证，而应使用 Validating 和 Validated
MouseDown	在鼠标指针指向一个控件，且鼠标按钮被按下时引发。这与 Click 事件不同，因为在按钮被按下之后，且未被释放之前引发 MouseDown
MouseMove	在鼠标滑过控件时引发



(续表)

事 件	说 明
MouseUp	在鼠标指针位于控件上，且鼠标按钮被释放时引发
Paint	绘制控件时引发
Validated	当控件的 CausesValidation 属性设置为 true，且该控件获得焦点时，引发该事件。它在 Validating 事件之后发生，表示验证已经完成
Validating	当控件的 CausesValidation 属性设置为 true，且该控件获得焦点时，引发该事件。注意，被验证的控件是正在失去焦点的控件，而不是正在获得焦点的控件

本章后面的示例将介绍上表中的许多事件。所有的示例都使用相同的格式，即首先创建窗体的可视化外观，选择并定位控件，再添加事件处理程序，事件处理程序包含了示例的主要工作代码。

有 3 种处理事件的基本方式。第一种是双击控件，进入控件默认事件的处理程序，这个事件因控件而异。如果该事件就是我们需要的事件，就可以开始编写代码。如果需要的事件与默认事件不同，有两种方法来处理这种情况。

一种方法是使用 Properties 窗口中的 Events 列表，单击如图 15-2 所示的闪电图标按钮，就会显示 Events 列表。

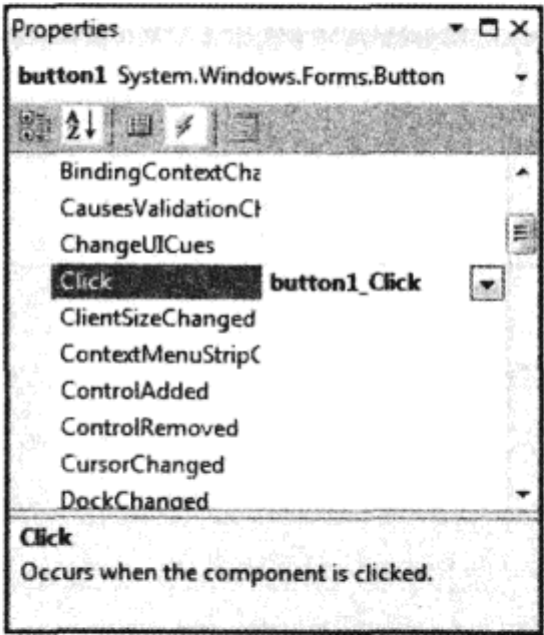


图 15-2

要给事件添加处理程序，只需在 Events 列表中双击该事件，就会生成给控件订阅该事件的代码，以及处理该事件的方法签名。另外，还可以在 Events 列表中该事件的旁边，为处理该事件的方法输入一个名称。按下回车键，就会用我们输入的名称生成一个事件处理程序。

另一个选项是自己添加订阅该事件的代码。在键入订阅该事件所需的代码时，VS 会检测到我们做的工作，并在代码中添加方法签名，就好像在窗体设计器中一样。

注意这两种方式都需要两步：订阅事件和处理方法的正确签名。如果双击控件，给要处理的事件编辑默认事件的方法签名，以处理另一个事件，就会失败，因为还需要修改 InitializeComponent() 中的事件订阅代码，所以这种方法并不是处理特定事件的快捷方式。

下面开始讨论控件本身，首先讨论 Windows 应用程序中最常用的一个控件，即 Button 控件。

## 15.2 Button 控件

在考虑按钮时，可能会把它想像为一个矩形按钮，单击该按钮，就可以执行某项任务。但.NET Framework 提供了一个派生于 Control 的类 System.Windows.Forms.ButtonBase，它实现了 Button 控件所需的基本功能，所以程序员可以从这个类中派生，创建定制的 Button 控件。

System.Windows.Forms 名称空间提供了 3 个派生于 ButtonBase 的控件，即 Button、CheckBox 和 RadioButton。本节主要讨论 Button 控件(这是标准的矩形按钮)，后面再介绍另外两个按钮。

Button 控件存在于几乎所有的 Windows 对话框中。按钮主要用于执行 3 类任务：

- 用某种状态关闭对话框(如 OK 和 Cancel 按钮)。
- 给对话框上输入的数据执行操作(例如，输入一些搜索条件后，单击 Search)。
- 打开另一个对话框或应用程序(如 Help 按钮)。

对 Button 控件的处理是非常简单的。通常是在窗体上添加控件，再双击它，给Click 事件添加代码，这对于大多数应用程序来说就足够了。

### 15.2.1 Button 控件的属性

下面介绍该控件的常用属性，了解该如何操作它。表 15-3 列出了 Button 类最常用的属性，但从技术上讲，它们都是在 ButtonBase 基类中定义的。这里只解释最常用的属性。完整的列表请参阅.NET Framework SDK 文档说明。

表 15-3

属 性	说 明
FlatStyle	可以用这个属性改变按钮的样式。如果把样式设置为 PopUp，则该按钮就显示为平面，直到用户再把鼠标指针移动到它上面为止。此时，按钮会弹出，显示为 3D 外观
Enabled	这个属性派生于 Control，但这里仍讨论它，因为这是一个非常重要的属性。把 Enabled 设置为 false，则该按钮就会灰显，单击它，不会起任何作用
Image	可以指定一个在按钮上显示的图像(位图，图标等)
ImageAlign	指定按钮上的图像在什么地方显示

### 15.2.2 Button 控件的事件

到目前为止，按钮最常用的事件是 Click。只要用户单击了按钮，即当鼠标指向该按钮时，按下鼠标左键，再释放它，就会引发该事件。如果在按钮上单击了鼠标左键，然后把鼠标移动到其他位置，再释放鼠标，将不会引发 Click 事件。同样，在按钮得到焦点，且用户按下了回车键时，也会引发 Click 事件。如果窗体上有一个按钮，就总是要处理这个事件。

在下面的示例中，创建一个带有 3 个按钮的对话框。其中两个按钮在英语和丹麦语之间来回切换(也可以使用其他语言)，最后一个按钮关闭对话框。

### 试一试：按钮的测试

按照下列步骤创建一个小型 Windows 应用程序，使用 3 个按钮来改变对话框标题的文本：

- (1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 ButtonDialog。
  - (2) 单击窗口右上角的 x 旁边的图钉图标，钉住工具箱，双击 Button 控件 3 次。然后移动按钮，重新设置窗体的大小，如图 15-3 所示。
  - (3) 右击一个按钮，选择 Properties，在 Properties 窗口上选择(Name)编辑字段，键入相关的文本，修改每个按钮的 Name 属性。
  - (4) 与 Name 字段一样，修改每个按钮的 Text 属性，但不修改 Text 属性值的 button 前缀。
  - (5) 我们要在文本的前面显示一个标志，清晰地表示出每个按钮的作用。选择 English 按钮，找到 Image 属性。单击右边的(...)，打开一个对话框，该对话框可以把图像添加到窗体的资源文件中。单击 Import 按钮，浏览图标。我们要显示的图标包含在 ButtonDialog 项目中。该项目可以从 Wrox 主页上下载。选择图标 UK.PNG 和 DK.PNG 文件。
  - (6) 选择 UK，单击 OK。然后选择 buttonDanish，单击 Image 属性上的(...)，选择 DK，再单击 OK。
  - (7) 注意按钮文本和图标彼此遮挡，所以需要改变图标的对齐方式。对于 English 和 Danish 按钮，把 ImageAlign 属性改为 MiddleLeft。
  - (8) 此时，可以调整按钮的宽度，使文本不会正好从图像的右边开头。为此，可以选择每个按钮，在文本和图像之间留出间距。
  - (9) 最后单击窗体，把 Text 属性改为 “Do you speak English?”。
- 这就是对话框的用户界面，如图 15-4 所示。

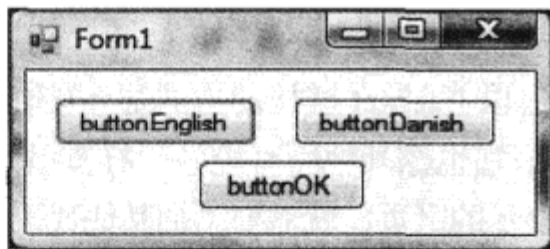


图 15-3

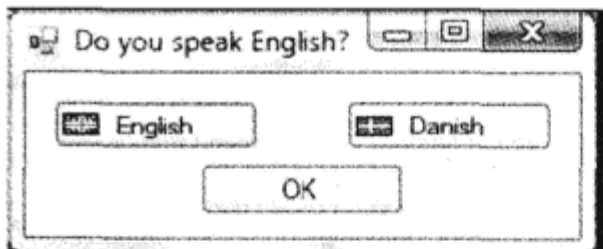


图 15-4

下面准备给对话框添加事件处理程序。双击 English 按钮，进入该控件默认事件的处理程序。Click 事件是按钮的默认事件，所以创建了它的处理程序。

#### 15.2.3 添加事件处理程序

双击 English 按钮，在事件处理程序中添加如下代码：

```
private void buttonEnglish_Click (object sender, EventArgs e)
{
    Text = "Do you speak English?";
}
```

在 Visual Studio 创建处理事件的方法时，方法名是控件名、下划线和要处理的事件名的组合。

对于 Click 事件，第一个参数 object sender 包含被单击的控件。在这个示例中，控件总是由方法名来标识，但在其他情况下，许多控件可能使用同一个方法来处理事件，此时就要通过查看这个值，

来确定是哪个控件调用了该方法。本章后面的“文本框控件”一节说明了多个控件如何使用同一个方法。另一个参数 `System.EventArgs` 包含实际发生的事情的信息。在本例中，不需要这些信息。

返回设计视图，双击 `Danish` 按钮，进入这个按钮的事件处理程序，下面是代码：

```
private void buttonDanish_Click(object sender, EventArgs e)
{
    Text = "Taler du dansk?";
}
```

这个方法与 `btnEnglish_Click` 相同，但文本是丹麦文字。最后，以相同的方式添加 `OK` 按钮的事件处理程序。其代码有一些不同之处：

```
private void buttonOK_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

使用这段代码，就可以退出应用程序。这就是第一个示例。编译这个示例，运行它，单击其中的几个按钮，会发现对话框标题栏上的文本改变了。

### 15.3 Label 和 LinkLabel 控件

`Label` 控件也许是最常用的控件。在任何 `Windows` 应用程序中，都可以在对话框中见到它们。标签是一个简单的控件，其用途只有一个：在窗体上显示文本。

`.NET Framework` 包含两个标签控件，它们可以用两种截然不同的方式来显示：

- `Label` 是标准的 `Windows` 标签。
- `LinkLabel` 类似于标准标签(派生于标准标签)，但以 `Internet` 链接的方式显示(超链接)。

标准的 `Label` 通常不需要添加任何事件处理代码。但它也像其他所有控件一样支持事件。对于 `LinkLabel` 控件，如果希望用户可以单击它，进入文本中显示的网页，就需要添加其他代码。

`Label` 控件有非常多的属性。大多数属性都派生于 `Control`，但有一些属性是新增的。表 15-4 列出了最常见的属性。如果没有特别说明，`Label` 和 `LinkLabel` 控件中都存在这些属性。

表 15-4

属 性	说 明
<code>BorderStyle</code>	可以指定标签边框的样式。默认为无边框
<code>FlatStyle</code>	控制显示控件的方式。把这个属性设置为 <code>PopUp</code> ，表示控件一直显示为平面样式，直到用户把鼠标指针移动到该控件上面，此时，控件显示为弹起样式
<code>Image</code>	指定要在标签上显示的图像(位图，图标等)
<code>ImageAlign</code>	指定图像显示在标签的什么地方
<code>LinkArea</code>	(只用于 <code>LinkLabel</code> )文本中显示为链接的部分
<code>LinkColor</code>	(只用于 <code>LinkLabel</code> )链接的颜色

(续表)

属 性	说 明
Links	(只用于 LinkLabel)LinkLabel 可以包含多个链接。利用这个属性可以查找需要的链接。控件会跟踪显示文本中的链接。不能在设计期间使用
LinkVisited	(只用于 LinkLabel)把它设置为 true，单击控件，链接就会显示为另一种颜色
TextAlign	指定文本显示在控件的什么地方
VisitedLinkColor	(只用于 LinkLabel)用户单击 LinkLabel 后控件的颜色

15.4 TextBox 控件

在希望用户输入程序员在设计阶段不知道的文本(如用户的姓名)时，应使用文本框。文本框的主要用途是让用户输入文本，用户可以输入任何字符，也可以只允许用户输入数值。

.NET Framework 内置了两个基本控件来提取用户输入的文本：TextBox 和 RichTextBox。这两个控件都派生于基类 TextBoxBase，而 TextBoxBase 派生于 Control。

TextBoxBase 提供了在文本框中处理文本的基本功能，例如选择文本、剪切和从剪切板上粘贴，以及许多事件。这里不讨论派生关系，而是先介绍两个控件中比较简单的一个：TextBox。下面创建一个示例，说明 TextBox 的属性，后面在此基础上说明 RichTextBox 控件。

15.4.1 TextBox 控件的属性

如本章前面所述，列出控件的所有属性是不可能的，所以表 15-5 仅列出最常见的属性。

表 15-5

属 性	说 明
CausesValidation	当控件的这个属性设置为 true，且该控件要获得焦点时，会引发两个事件：Validating 和 Validated。可以处理这些事件，以便验证正在失去焦点的控件中数据的有效性。这可能使控件永远都不能获得焦点。下一节会讨论相关的事件
CharacterCasing	这个值表示 TextBox 是否会改变输入的文本的大小写。可能的值有： <ul style="list-style-type: none"><li>• Lower: 输入的所有文本都转换为小写</li><li>• Normal: 不对文本进行任何转换</li><li>• Upper: 输入的所有文本都转换为大写</li></ul>
MaxLength	这个值指定输入到 TextBox 中的文本的最大字符长度。把这个值设置为 0，表示最大字符长度仅受限于可用的内存
Multiline	表示该控件是否是一个多行控件。多行控件可以显示多行文本。如果将 Multiline 属性设置为 true，通常也把 WordWrap 也设置为 true
PasswordChar	指定是否用密码字符替换在单行文本框中输入的字符。如果 Multiline 属性为 true，这个属性就不起作用
ReadOnly	这个 Boolean 值表示文本是否为只读

(续表)

属 性	说 明
ScrollBars	指定多行文本框是否显示滚动条
SelectedText	在文本框中选择的文本
SelectionLength	在文本中选择的字符数。如果这个值设置得比文本中的总字符数大，则控件会把它重新设置为字符总数减去 SelectionStart 的值
SelectionStart	文本框中被选中文本的开头
WordWrap	指定在多行文本框中，如果一行的宽度超出了控件的宽度，其文本是否应自动换行

15.4.2 TextBox 控件的事件

在窗体上，对 TextBox 控件中文本进行精细的有效性验证会使一些用户很高兴，使另一些用户则会感到很生气。当用户单击了 OK 按钮后，对话框只验证其内容，此时用户可能非常生气。这种验证数据有效性的方式通常会显示一个信息框，告诉用户“第三个 TextBox”中的数据不正确。接着继续单击 OK 按钮，直到所有的数据都正确为止。显然这不是验证数据有效性的好方法，那么我们还能怎么做呢？

答案取决于 TextBox 控件提供的有效性验证事件。如果要确保文本框中不输入无效的字符，或者只输入某个范围内的数值，就需要告诉控件的用户：输入的值是否有效。

TextBox 控件提供了表 15-6 所示的事件(所有的事件都派生于 Control)。

表 15-6

名 称	说 明
Enter Leave Validating Validated	这 4 个事件按照列出的顺序引发。它们统称为“焦点事件”，当控件的焦点发生改变时引发，但有两个例外。Validating 和 Validated 仅在控件接收了焦点，且其 CausesValidation 属性设置为 true 时引发。接收焦点的控件引发事件的原因是有时即使焦点改变了，我们也不希望验证控件的有效性。例如用户单击了 Help 按钮
KeyDown KeyPress KeyUp	这 3 个事件称为“键事件”。它们可以监视和改变输入到控件中的内容。KeyDown 和 KeyUp 接收与所按下键对应的键码，这样就可以确定是否按下了特殊的键 Shift 或 Ctrl 和 F1。另一方面，KeyPress 接收与键对应的字符。这表示字母 a 的值与字母 A 的值不同。如果要排除某个范围内的字符，例如只允许输入数值，这是很有用的
TextChanged	只要文本框中的文本发生了改变，无论发生什么改变，都会引发该事件

下面的示例将创建一个对话框，在该对话框中可以输入姓名、地址、职业和年龄。这个示例的目的是为处理属性和使用事件打下基础，而不是创建什么特别有用的东西。

试一试：TextBoxTest

先建立用户界面：

- (1) 在 C:\BegVCS\Chapter15 目录中创建一个新的 Windows 应用程序 TextBoxControls。



(2) 创建如图 15-5 所示的窗体，把标签、文本框和按钮拖放到设计界面上。在重新设置两个文本框 `textBoxAddress` 和 `textBoxOutput` 的大小时，必须把它们的 `Multiline` 属性设置为 `true`。为此，右击控件，选择 `Properties`。

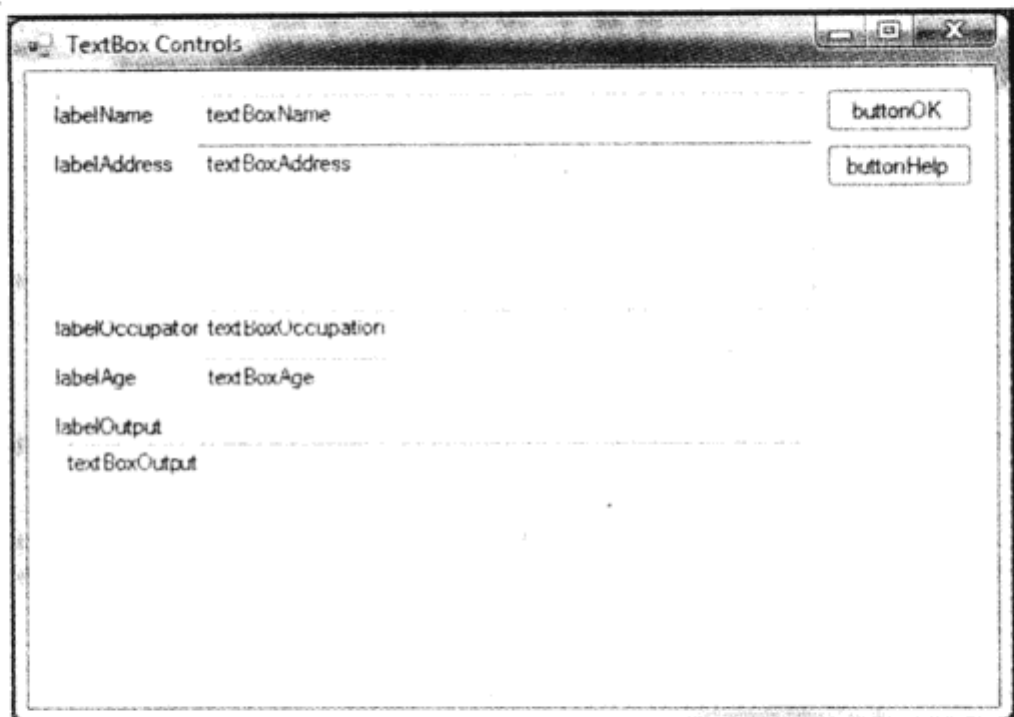


图 15-5

- (3) 给控件命名，如图 15-5 所示。
- (4) 把其他控件的 `Text` 属性设置为控件的名称，但不改变表示控件类型的前缀(即 `Button` 和 `TextBox` 和 `Label`)。把窗体的 `Text` 属性设置为 `TextBoxControls`。
- (5) 把两个控件 `textBoxOutput` 和 `textBoxAddress` 的 `Scrollbars` 属性设置为 `Vertical`。
- (6) 把 `textBoxOutput` 控件的 `ReadOnly` 属性设置为 `true`。
- (7) 把按钮 `btnHelpButton` 的 `CausesValidation` 属性设置为 `false`。在前面讨论 `Validating` 和 `Validated` 事件时，把这个属性设置为 `false`，就可以让用户单击这个按钮，而不必考虑输入的无效数据。
- (8) 改变窗体的大小，使之适合于控件的大小，然后锚定控件，这样它们就可以在重新设置窗体的大小时正确地响应。下面一次设置每类控件的 `Anchor` 属性。首先，按住 `Ctrl` 键，依次选择除了 `textBoxOutput` 之外的所有文本框控件。然后在 `Properties` 窗口中，把 `Anchor` 属性设置为 `Top, Left, Right`，这就为每个选中的文本框控件设置了 `Anchor` 属性。再选择 `textBoxOutput` 控件，把 `Anchor` 属性设置为 `Top, Bottom, Left, Right`。现在把两个按钮控件的 `Anchor` 属性设置为 `Top, Right`。
- 锚定 `textBoxOutput` 而不是让它停靠在窗体底部的原因是，在拖动窗体时，要重新设置输出文本区域的大小。如果把该控件停靠在窗体的底部，它就会随窗体一起移动，但不会重新设置大小。
- (9) 最后要设置的是，在窗体上，找到 `Size` 和 `MinimumSize` 属性。如果窗体设置得比现在的小，就没有什么意义了，因此应把 `MinimumSize` 属性值设置得与 `Size` 属性值一样大。

#### 示例的说明

设置窗体的可见部分现在已经完成了。如果运行它，则单击按钮或输入文本，将不会发生什么情况。但如果最大化或拖动对话框，控件就会按照希望的那样在用户界面上位于正确的位置，并重新设置其大小，以填充整个对话框。



## 15.4.3 添加事件处理程序

在设计视图中, 双击 buttonOK 按钮, 对另一个按钮重复这个过程。与本章前面的按钮示例一样, 这会创建按钮的 Click 事件处理程序。单击 OK 按钮, 把输入文本框中的文本传送到只读的输出框中。

下面是两个 Click 事件处理程序的代码:



```
private void buttonOK_Click(object sender, EventArgs e)
{
    // No testing for invalid values are made, as that should
    // not be necessary

    string output;

    // Concatenate the text values of the four TextBoxes.
    output = "Name: " + this.textBoxName.Text + "\r\n";
    output += "Address: " + this.textBoxAddress.Text + "\r\n";
    output += "Occupation: " + this.textBoxOccupation.Text + "\r\n";
    output += "Age: " + this.textBoxAge.Text;

    // Insert the new text.
    this.textBoxOutput.Text = output;
}

private void buttonHelp_Click(object sender, EventArgs e)
{
    // Write a short description of each TextBox in the Output TextBox.
    string output;

    output = "Name = Your name\r\n";
    output += "Address = Your address\r\n";
    output += "Occupation = Only allowed value is 'Programmer'\r\n";
    output += "Age = Your age";

    // Insert the new text.
    this.textBoxOutput.Text = output;
}
```

代码段 Chapter15\TextBoxControls\Form1.cs

在这两个函数中, 使用了文本框的 Text 属性。textBoxAge 控件的 Text 属性获取输入的值, 作为人的年龄, textBoxOutput 控件的 text 属性用于显示串在一起的文本。

我们插入用户输入的信息, 无需检查该信息是否正确。这就是说, 必须在其他地方进行检查。在本例中, 必须满足许多条件, 其值才是正确的。

- 用户名不能为空。
- 用户的年龄必须是一个大于或等于 0 的数字。
- 用户的职业必须是“程序员”或为空。
- 用户的地址不能为空。

从中可以看出, 对两个文本框(textBoxName 和 textBoxAddress)进行的检查是相同的。并应禁止用户在 Age 框中输入无效的数值, 最后必须检查用户是不是程序员。

为了防止用户在输入完信息之前单击 OK 按钮，要先把 OK 按钮的 Enabled 属性设置为 false，这应在窗体的构造函数中设置，而不是在 Properties 窗口中设置。如果在构造函数中设置属性，应确保在调用 InitializeComponent() 中生成的代码之后，再设置这些属性：

```
public Form1()
{
    InitializeComponent();
    buttonOK.Enabled = false;
}
```

下面创建这两个文本框的处理程序，检查一下它们是否为空。为此，订阅文本框的 Validating 事件。因为需要在这两个控件上执行相同的操作，所以把同一个事件处理程序赋予它们。在窗体上选择这两个控件，再在 Events 列表中选择 Validating 事件，键入 textBoxEmpty\_Validating 作为事件名。单击闪电按钮，就可以在 Properties 窗口中打开 Events 列表。

与前面的按钮事件处理程序不同，Validating 事件的处理程序是标准处理程序 System.EventHandler 的一个专用版本。这个事件需要专用处理程序的原因是，如果有效性验证失败，就必须有一种方式防止进行任何进一步的处理。如果要取消进一步的处理，就表示在输入有效的数据前，不能退出该文本框。

在以前的 Visual Studio 版本中，使用 GotFocus 和 LostFocus 事件执行控件的有效性验证时，Validating、Validated 事件和 CausesValidation 属性一起更正了一个错误。当 GotFocus 和 LostFocus 事件被连续引发时，就产生了这个错误；因为有效性验证代码试图在控件之间移动焦点，这将产生一个无限循环。

用下面的代码替换 VS 在事件处理程序中生成的 throw 语句：

```
private void textBoxEmpty_Validating (object sender,
                                     System.ComponentModel.CancelEventArgs e)
{
    TextBox tb = (TextBox)sender;

    if (tb.Text.Length == 0)
        tb.BackColor = Color.Red;
    else
        tb.BackColor = System.Drawing.SystemColors.Window;
    ValidateOK();
}
```

因为有多文本框使用这个方法来处理事件，所以我们不知道哪个控件调用了函数，但无论是哪个控件调用了方法，其结果是一样的，所以可以对传送给文本框的 sender 参数进行类型转换，对它执行操作。

```
TextBox tb = (TextBox)sender;
```

如果文本框中的文本长度是 0，就把背景色设置为红色，把 Tag 设置为 false。如果不是，就把背景色设置为窗口的标准 Windows 颜色。



在设置控件的标准颜色时，应总是使用 `System.Drawing.SystemColors` 枚举中的颜色。如果把颜色设置为白色，而用户修改了默认的颜色设置，应用程序看起来就会很奇怪。

`ValidateOK()`函数将在本例的最后介绍。与 `Validating` 事件相对照，下一个添加的处理程序是 `Occupation` 文本框的处理程序。这个过程与前面两个处理程序完全相同，但有效性验证代码有所不同，因为职业必须是 `Programmer` 或一个空字符串。要添加事件处理程序，应双击 `textBoxOccupation` 控件的 `Validating` 事件。

然后添加处理程序本身：

```
private void textBoxOccupation_Validating(object sender,
                                         System.ComponentModel.CancelEventArgs e)
{
    TextBox tb = (TextBox)sender;

    if (tb.Text == "Programmer" || tb.Text.Length == 0)
        tb.BackColor = System.Drawing.SystemColors.Window;
    else
        tb.BackColor = Color.Red;
    ValidateOK();
}
```

倒数第二个要处理的事项是处理 `Age` 文本框。我们希望用户只键入正数(包括 0，这样可以使检测简单一些)。为此，使用 `KeyPress` 事件，在不想要的字符在文本框中显示出来之前就删除它们。还要把输入到控件中的字符数限制为 3 个。

首先，把 `textBoxAge` 控件的 `MaxLength` 属性设置为 3，然后在 `Properties` 窗口的 `Events` 列表中双击 `KeyPress` 事件，以订阅它。`KeyPress` 事件处理程序也是专用的，其中提供了 `System.Windows.Forms.KeyPressEventHandler`，因为事件需要被按下键的信息。

然后给事件处理程序添加如下代码：

```
private void textBoxAge_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8)
        e.Handled = true;
}
```

0~9 之间数字的 ASCII 值是 48~57，所以应保证字符在这个范围内。但有一个例外。ASCII 值 8 表示退格键，为了编辑方便，允许跳过它。把 `KeyPressEventArgs` 的 `Handled` 属性设置为 `true`，告诉控件不对字符进行其他任何操作，所以如果按下的键不是数字或退格，就不显示该字符。

现在控件没有标记为有效或无效，这是因为需要进行另一个检查，看看是否输入了信息。这是很简单的，因为前面已经编写了执行该检查的方法，在 `Events` 列表中给 `textBoxAge` 控件选择 `Validating` 事件下拉列表中的 `textBoxEmpty_Validating` 事件处理程序。

最后一件事：启用或禁用 OK 按钮的 `ValidateOK` 方法：

```
private void ValidateOK()
```

```

{
    buttonOK.Enabled = (textBoxName.BackColor != Color.Red &&
        textBoxAddress.BackColor != Color.Red &&
        textBoxOccupation.BackColor != Color.Red &&
        textBoxAge.BackColor != Color.Red);
}

```

如果所有的文本框都不使用红色作为背景色，这个方法就把 OK 按钮的 Enabled 属性设置为 true。

如果现在测试该程序，就会得到如图 15-6 所示的结果。注意，在文本框中输入了无效的数据，但背景色没有改为红色时，可以单击 Help 按钮。

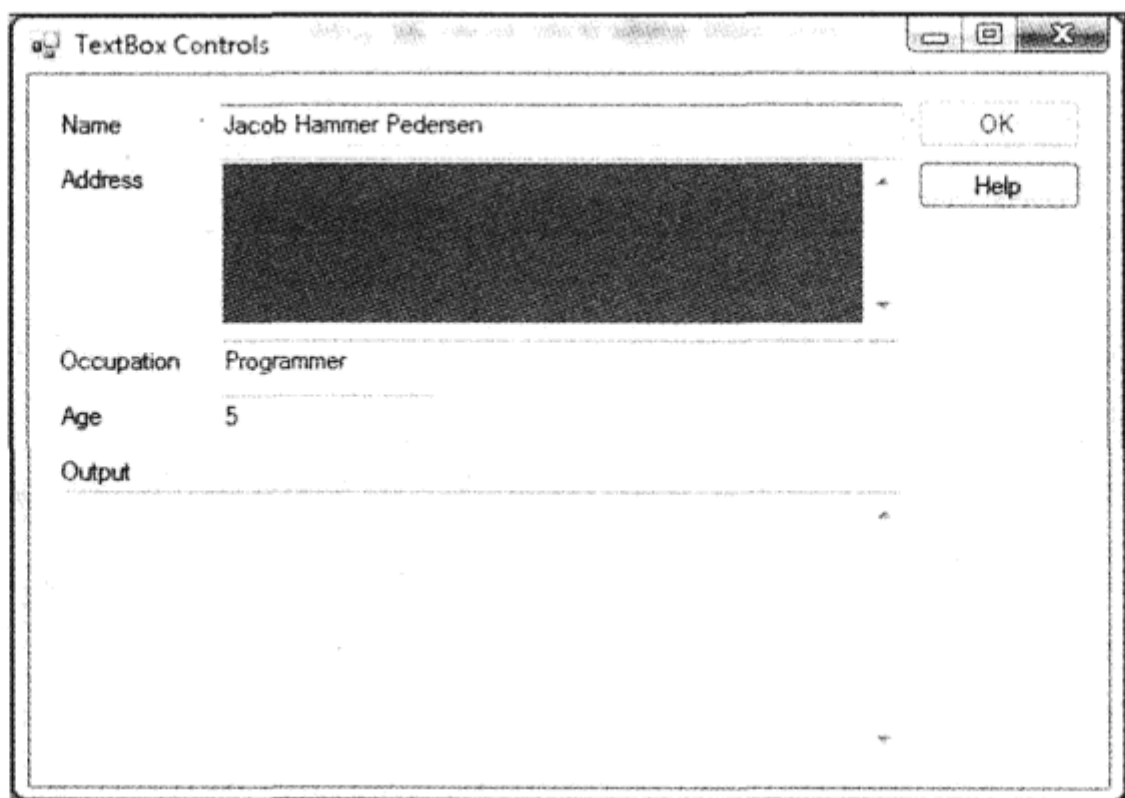


图 15-6

## 15.5 RadioButton 和 CheckBox 控件

如前所述，RadioButton 和 CheckBox 控件与 Button 控件有相同的基类，但它们的外观和用法大不相同。

传统上，单选按钮显示为一个标签，左边是一个圆点，该点可以是选中或未选中。在要给用户提供两个或多个互斥选项时，就可以使用单选按钮。例如，询问用户的性别。

把单选按钮组合在一起，给它们创建一个逻辑单元，此时必须使用 GroupBox 控件或其他一些容器。首先在窗体上拖放一个组框，再把需要的 RadioButton 按钮放在组框的边界之内，RadioButton 按钮会自动改变自己的状态，以使组框中只选中一个选项。如果不把它们放在组框中，则在任意时刻，窗体上只有一个 RadioButton 被选中。

传统上，CheckBox 显示为一个标签，左边是一个小方框。在希望用户可以选择一个或多个选项时，就应使用复选框。例如询问用户要使用的操作系统(如 Windows Vista、Windows XP 和 Linux 等)。

下面介绍这两个控件的重要属性和事件，从 RadioButton 开始，然后用一个简短小示例说明它们的用法。

15.5.1 RadioButton 控件的属性

这个控件派生于 `ButtonBase`，前面已经有一个使用按钮的示例了，所以需要描述的属性仅有几个，如表 15-7 所示。完整的列表请参阅 .NET Framework SDK 文档说明。

表 15-7

属 性	说 明
Appearance	RadioButton 可以显示为一个标签，相应的圆点放在左边、中间或右边，或者显示为标准按钮。当它显示为按钮时，控件被选中时显示为按下状态，否则显示为弹起状态
AutoCheck	如果这个属性为 <code>true</code> ，用户单击单选按钮时，会显示一个选中标记。如果该属性为 <code>false</code> ，就必须在 <code>Click</code> 事件处理程序的代码中手工选中单选按钮
CheckAlign	使用这个属性，可以改变单选按钮的复选框的对齐形式，默认是 <code>ContentAlignment.MiddleLeft</code>
Checked	表示控件的状态。如果控件有一个选中标记，它就是 <code>true</code> ，否则为 <code>false</code>

15.5.2 RadioButton 控件的事件

在处理 `RadioButton` 控件时，通常只使用一个事件，但还可以订阅许多其他事件。本章只介绍两个事件，介绍第二个事件的原因是它们之间有微妙的区别，如表 15-8 所示。

表 15-8

属 性	说 明
CheckedChanged	当 <code>RadioButton</code> 的选中选项发生改变时，引发这个事件
Click	每次单击 <code>RadioButton</code> 时，都会引发该事件。这与 <code>CheckedChange</code> 事件是不同的，因为连续单击 <code>RadioButton</code> 两次或多次只改变 <code>Checked</code> 属性一次(而且只有尚未选中时才如此)。而且，如果被单击按钮的 <code>AutoCheck</code> 属性是 <code>false</code> ，则该按钮根本不会被选中，只引发 <code>Click</code> 事件

15.5.3 CheckBox 控件的属性

可以想像，这个控件的属性和事件非常类似于 `RadioButton` 控件，但有两个新属性，如表 15-9 所示。

表 15-9

属 性	说 明
CheckState	与 <code>RadioButton</code> 不同， <code>CheckBox</code> 有 3 种状态： <code>Checked</code> 、 <code>Indeterminate</code> 和 <code>Unchecked</code> 。复选框的状态是 <code>Indeterminate</code> 时，控件旁边的复选框通常是灰色的，表示复选框的当前值是无效的。或者无法确定(例如，如果选中标记表示文件的只读状态，且选中了两个文件，则其中一个文件是只读的，另一个文件不是)，或者在当前环境下没有意义
ThreeState	这个属性为 <code>false</code> 时，用户就不能把 <code>CheckState</code> 属性改为 <code>Indeterminate</code> 。但仍可以在代码中把 <code>CheckState</code> 属性改为 <code>Indeterminate</code>

15.5.4 CheckBox 控件的事件

一般只使用这个控件的一两个事件。注意，RadioButton 和 CheckBox 控件都有 CheckChanged 事件，但效果不同，如表 15-10 所示。

表 15-10

事 件	说 明
CheckedChanged	当复选框的 Checked 属性发生改变时，就引发该事件。注意在复选框中，当 ThreeState 属性为 true 时，单击复选框可能不会改变 Checked 属性。在复选框从 Checked 变为 Indeterminate 状态时，就会出现这种情况
CheckedStateChanged	当 CheckedState 属性改变时，引发该事件。CheckedState 属性的值可以是 Checked 和 Unchecked。只要 Checked 属性改变了，就引发该事件。另外，当状态从 Checked 变为 Indeterminate 时，也会引发该事件

前面总结了 RadioButton 和 CheckBox 控件的事件和属性。在使用它们之前，先介绍一下前面提及的 GroupBox 控件。

15.5.5 GroupBox 控件

GroupBox 控件常常用于合理地组合一组控件，如 RadioButton 及 CheckBox 控件，显示一个框架，其上有一个标题。

组框的用法非常简单，把它拖放到窗体上，再把所需的控件拖放到组框中即可(但其顺序不能颠倒——不能把组框放在已有的控件上面)。其结果是父控件是组框，而不是窗体，所以在任意时刻，可以选择多个 RadioButton。但在组框中，一次只能选择一个 RadioButton。

这里需要解释一下父控件和子控件的关系。把一个控件放在窗体上时，窗体就是该控件的父控件，所以该控件是窗体的一个子控件。而把一个GroupBox 放在窗体上时，它就成为窗体的一个子控件。而组框本身可以包含控件，所以它就是这些控件的父控件，其结果是移动 GroupBox 时，其中的所有控件也会随之移动。

把控件放在组框上的另一个结果是可以改变其中所有控件的某些属性，方法是在组框上设置这些属性。例如，如果要禁用组框中的所有控件，只需把组框的 Enabled 属性设置为 false 即可。

下面用一个示例说明 GroupBox 控件的用法。

试一试：RadioButton 和 CheckBox 示例

下面修改本章全面创建的 TextBoxControls 示例。在该示例中，唯一可能的职业是程序员。下面不强迫用户填写程序员，而是把这个文本框改成复选框。为了说明 RadioButton 的用法，我们将要求用户再提供一条信息：性别。

把文本框示例改为：

- (1) 删除 labelOccupation 标签和文本框 textBoxOccupation。
- (2) 添加一个CheckBox、一个 GroupBox 和两个 RadioButton 控件，并命名这些新控件，如图 15-7 所示。注意与前面使用的其他控件不同，GroupBox 控件位于Toolbox 面板的 Containers 选项卡上。



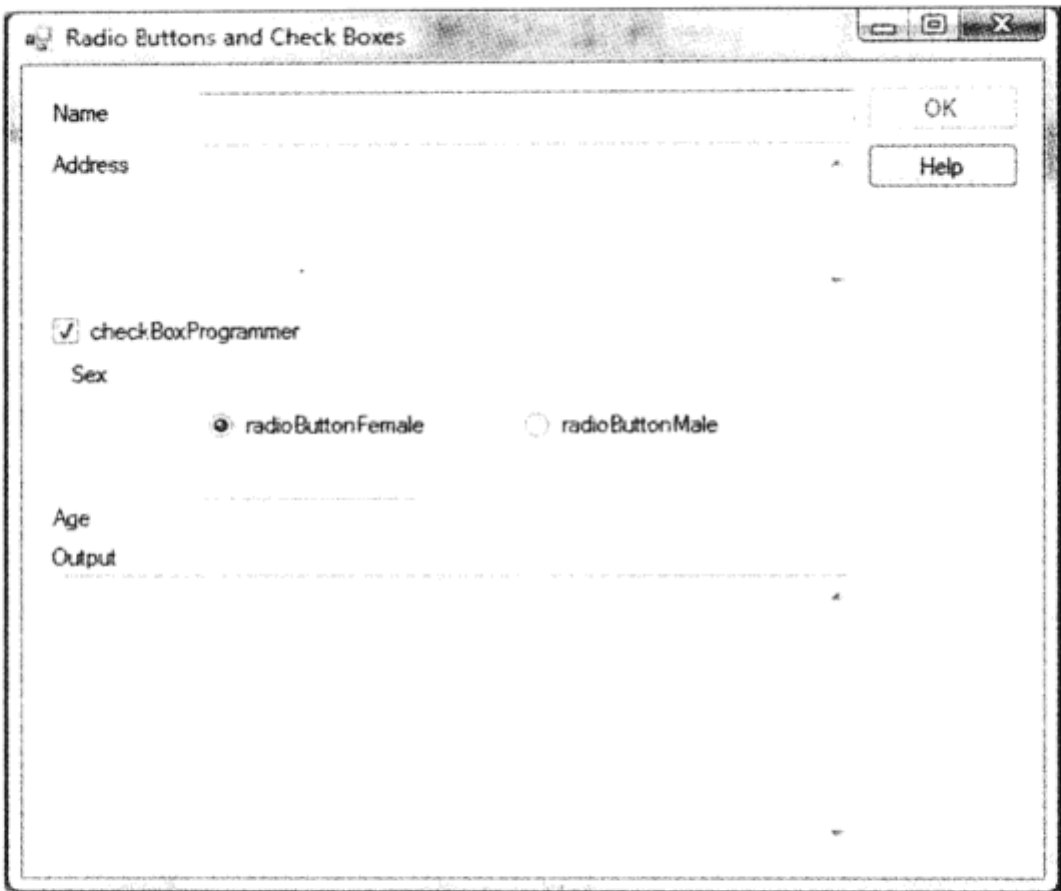


图 15-7

(3) RadioButton 和 CheckBox 控件的 Text 属性应与该控件名相同(前 3 个字符不算)。GroupBox 的 Text 属性应是 Sex。

(4) 把 checkBoxProgrammer 复选框的 Checked 属性设置为 true。注意 CheckState 属性自动改为 Checked。

(5) 把 radioButtonMale 或 radioButtonFemale 的 Checked 属性设置为 true。注意不能把它们两个同时设置为 true。否则，另一个 RadioButton 的值会自动变为 false。

对这个示例的可见部分不再需要更多的修改，但代码要进行许多修改。首先，需要删除所有对已删除文本框的引用。进入代码，完成下述步骤。

(1) 在 ValidateOK 方法中，删除对 textBoxOccupation 背景的测试：



可从  
wrox.com  
下载源代码

```
private void ValidateOK()
{
    // Set the OK button to enabled if all the Tags are true.
    buttonOK.Enabled = (textBoxName.BackColor != Color.Red &&
        textBoxAddress.BackColor != Color.Red &&
        textBoxAge.BackColor != Color.Red);
}
```

代码段 Chapter15\Radio and Check Buttons\Form1.cs

(2) 彻底删除 textBoxOccupation\_Validating 方法。

(3) 删除 buttonOK\_Click 中的引用。

示例的说明

这里使用的是复选框，而不是文本框，所以用户不会输入无效的信息，因为用户要么是一个程序员，要么不是。



我们也知道用户要么是男性，要么是女性，因为前面把一个 `RadioButton` 的属性设置为 `true`，这样用户就不会选择无效的值。因此，下面只需要修改帮助文本和输出。在按钮事件处理程序中完成它：

```
private void buttonHelp_Click(object sender, System.EventArgs e)
{
    // Write a short description of each TextBox in the Output TextBox.
    string output;

    output = "Name = Your name\r\n";
    output += "Address = Your address\r\n";
    output += "Programmer = Check 'Programmer' if you are a programmer\r\n";
    output += "Sex = Choose your sex\r\n";
    output += "Age = Your age";

    // Insert the new text.
    this.textBoxOutput.Text = output;
}
```

只是修改了帮助文本，所以不要对 `help` 方法感到惊讶。OK 方法显得更有趣一点：

```
private void buttonOK_Click(object sender, System.EventArgs e)
{
    // No testing for invalid values are made, as that should
    // not be necessary

    string output;

    // Concatenate the text values of the four TextBoxes.
    output = "Name: " + this.textBoxName.Text + "\r\n";
    output += "Address: " + this.textBoxAddress.Text + "\r\n";
    output += "Occupation: " + (string)(this.chkProgrammer.Checked ?
        "Programmer" : "Not a programmer") + "\r\n";
    output += "Sex: " + (string)(this.radioButtonFemale.Checked ? "Female" :
        "Male") + "\r\n";
    output += "Age: " + this.textBoxAge.Text;

    // Insert the new text.
    this.textBoxOutput.Text = output;
}
```

在突出显示的代码中，第一行打印出了用户的职业。考察一下复选框的 `Checked` 属性，如果它是 `true`，就写入字符串“Programmer”，如果它是 `false`，就填写“Not a programmer”。

第二行代码检查单选按钮 `radioButtonFemale`。如果该控件的 `Checked` 属性是 `true`，则该用户是一位女性。如果它是 `false`，则该用户是一位男性。在启动程序时，可以不选中任何单选按钮，但因为是在设计期间选择了其中一个单选按钮，所以可以肯定总是会选中两个单选按钮中的一个。

现在运行示例，得到如图 15-8 所示的结果。

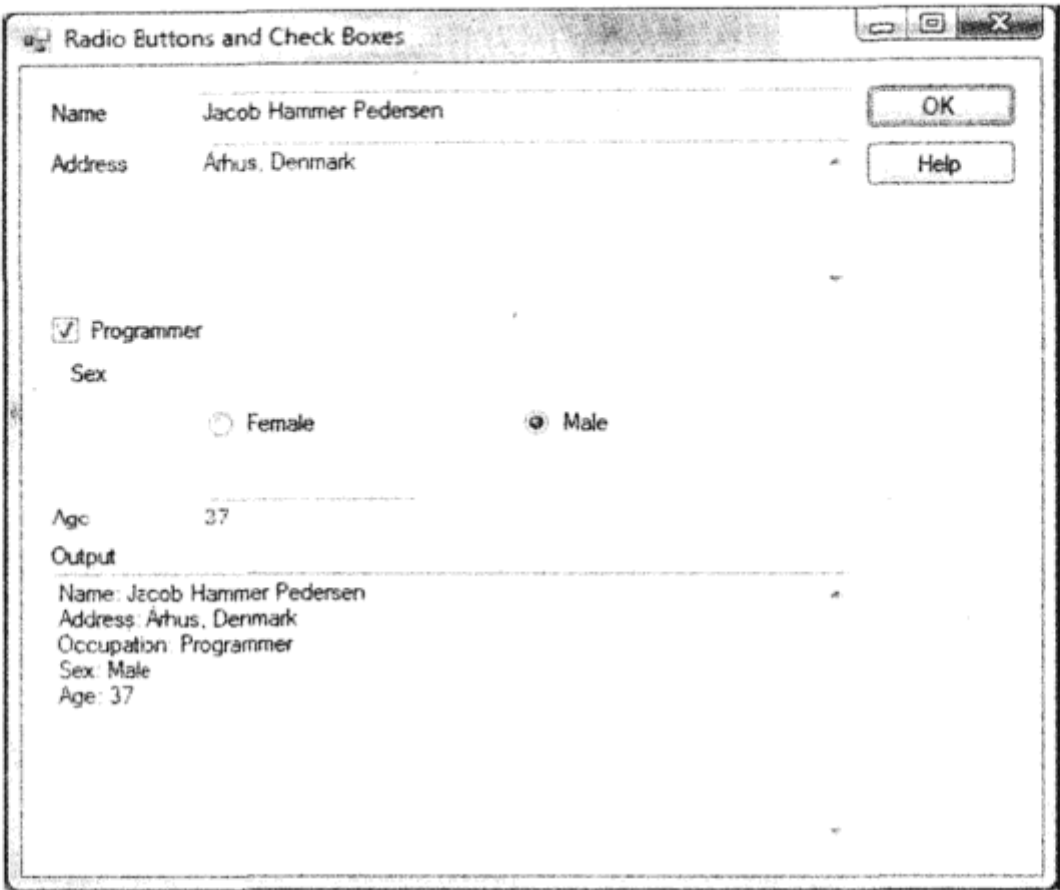


图 15-8

## 15.6 RichTextBox 控件

与常用的 `TextBox` 一样，`RichTextBox` 控件派生于 `TextBoxBase`。所以，它与 `TextBox` 共享许多功能，但许多功能是不同的。`TextBox` 常用于从用户处获取简短的文本字符串，而 `RichTextBox` 用于显示和输入格式化的文本(例如，黑体、下划线和斜体)。它使用标准的格式化文本，称为 `Rich Text Format` (富文本格式)或 `RTF`。

在上面的示例中，我们使用了标准的 `TextBox`。也可以使用 `RichTextBox` 来完成该任务。实际上，如后面的示例所示，可以删除 `textBoxOutput` 文本框，在它的位置上插入一个同名的 `RichTextBox`，这个示例还会像以前那样运行。

### 15.6.1 RichTextBox 控件的属性

如果这种文本框比上一节介绍的文本框更高级，我们就会期望它有一些新属性。表15-11 中列出了 `RichTextBox` 的一些最常用属性。

表 15-11

属 性	说 明
CanRedo	如果上一个被撤消的操作可以使用 Redo 重复，这个属性就是 true
CanUndo	如果可以在 RichTextBox 上撤消上一个操作，这个属性就是 true，注意，CanUndo 在 TextBoxBase 中定义，所以也可用于 TextBox 控件
RedoActionName	这个属性包含通过 Redo 方法执行的操作名称

(续表)

属 性	说 明
DetectUrls	把这个属性设置为 true, 可以使控件检测 URL, 并格式化它们(像在浏览器中那样有下划线)
Rtf	它对应于 Text 属性, 但包含 RTF 格式的文本
SelectedRtf	使用这个属性可以获取或设置控件中被选中的 RTF 格式文本。如果把相应文本复制到另一个应用程序中, 例如 Word, 该文本会保留所有的格式化信息
SelectedText	与 SelectedRtf 一样, 可以使用这个属性获取或设置被选中的文本。但与该属性的 RTF 版本不同, 所有的格式化信息都会丢失
SelectionAlignment	它表示选中文本的对齐方式, 可以是 Center、Left 或 Right
SelectionBullet	使用这个属性可以确定选中的文本是否格式化为项目符号的格式, 或使用它插入或删除项目符号
BulletIndent	使用这个属性可以指定项目符号的缩进像素值
SelectionColor	这个属性可以修改选中文本的颜色
SelectionFont	这个属性可以修改选中文本的字体
SelectionLength	使用这个属性可以设置或获取选中文本的长度
SelectionType	这个属性包含了选中文本的信息。它可以确定是选择了一个或多个 OLE 对象, 还是仅选择了文本
ShowSelectionMargin	如果把这个属性设置为 true, 在 RichTextBox 的左边就会出现页边距, 这将使用户更易于选择文本
UndoActionName	如果用户选择撤消某个动作, 该属性将获取该操作的名称
SelectionProtected	把这个属性设置为 true, 可以指定不修改文本的某些部分

从上表可以看出, 大多数新属性都与选中的文本有关。这是因为在用户处理其文本时, 对它们应用的任何格式化操作都是对用户选择出来的文本进行的。万一没有选择出文本, 格式化操作就从文本中光标所在的位置开始应用, 该位置称为插入点。

15.6.2 RichTextBox 控件的事件

RichTextBox 使用的大多数事件与 TextBox 使用的事件相同, 表 15-12 列出几个有趣的新事件。

表 15-12

名 称	说 明
LinkClicked	在用户单击文本中的链接时, 引发该事件
Protected	在用户尝试修改已经标记为受保护的文本时, 引发该事件
SelectionChanged	在选中文本发生变化时, 引发该事件。如果因某些原因不希望用户修改选中的文本, 就可以在这里禁止修改

在下面的示例中, 将创建一个非常基本的文本编辑器。它说明了如何修改文本的基本格式, 如

何加载和保存 RichTextBox 中的文本。为了简单起见，这个示例从固定文件中加载并保存在固定的文件中。

试一试：RichTextBox 示例

与往常一样，首先设计窗体：

- (1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 C# Windows 应用程序，命名为 Simple Text Editor。
- (2) 创建窗体，如图 15-9 所示。文本框 textBoxSize 应是一个 TextBox 控件。RichTextBoxText 文本框应是一个 RichTextBox 控件。

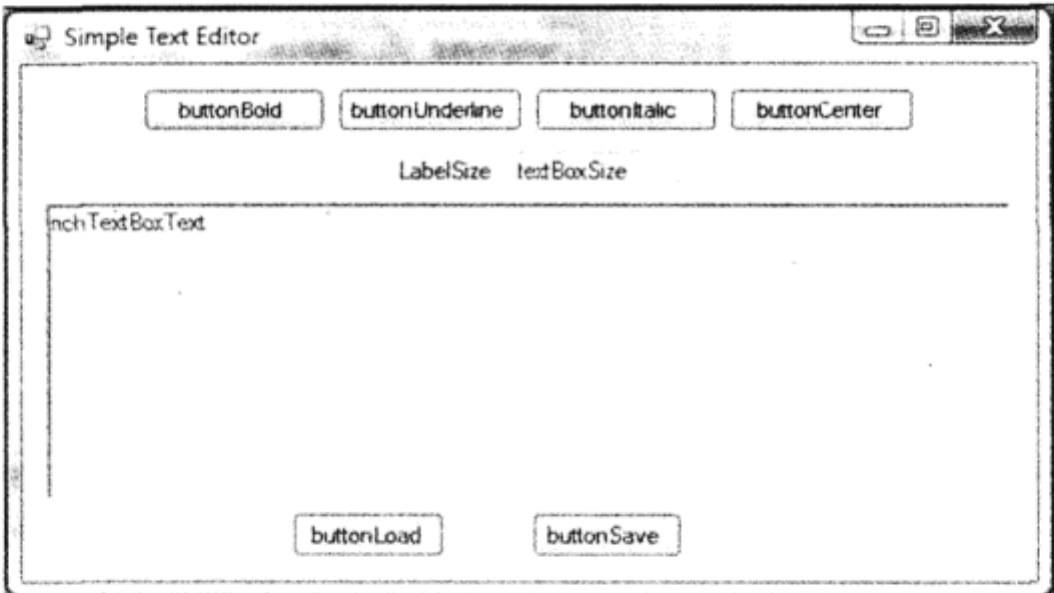


图 15-9

- (3) 如图 15-9 所示命名控件。
- (4) 除了文本框以外，把其他控件的 Text 属性设置为其控件名称(但表示该控件类型的名称的第一部分不算)。
- (5) 把 textBoxSize 文本框的 Text 属性改为 10。
- (6) 锚定控件，如表 15-13 所示。


表 15-13

控 件 名 称	Anchor 值
buttonLoad 和 buttonSave	Bottom
richTextBoxText	Top, Left, Bottom, Right
其他控件	Top

- (7) 把窗体的 MinimumSize 属性值设置为 Size 属性的值。

示例的说明

前面是该示例的可见部分，下面将分析代码。双击 Bold 按钮，在代码中添加 Click 事件处理程序。下面是该事件的代码：



可从  
wrox.com  
下载源代码

```
private void buttonBold_Click(object sender, EventArgs e)
{
    Font oldFont;
```

```

Font newFont;

oldFont = this.richTextBoxText.SelectionFont;

if (oldFont.Bold)
    newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
else
    newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

this.richTextBoxText.SelectionFont = newFont;
this.richTextBoxText.Focus();
}

```

代码段 Chapter15\Simple Text Editor\Form1.cs

首先获取当前选中文本使用的字体，并把它赋给一个局部变量 `oldFont`。然后检查一下选中文本是否为粗体。如果是，就去除粗体设置；否则就设置粗体。使用 `oldFont` 作为原型，创建一个新字体，但根据需要添加或删除粗体格式。

最后，把新字体赋给选中的文本，把焦点返回给 `RichTextBox`。

`buttonItalic` 和 `buttonUnderline` 的事件处理程序的代码与上面的代码相同，但检查对应样式的代码不同。双击 `Italic` 和 `Underline` 两个按钮，添加下列代码：

```

private void buttonItalic_Click(object sender, EventArgs e)
{
    Font oldFont;
    Font newFont;

    // Get the font that is being used in the selected text.
    oldFont = this.richTextBoxText.SelectionFont;

    // If the font is using Italic style now, we should remove it.
    if (oldFont.Italic)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
    else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

    // Insert the new font.
    this.richTextBoxText.SelectionFont = newFont;
    this.richTextBoxText.Focus();
}

private void buttonUnderline_Click(object sender, System.EventArgs e)
{
    Font oldFont;
    Font newFont;

    // Get the font that is being used in the selected text.
    oldFont = this.richTextBoxText.SelectionFont;

    // If the font is using Underline style now, we should remove it.
    if (oldFont.Underline)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
}

```

```

else
    newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

// Insert the new font.
this.richTextBoxText.SelectionFont = newFont;
this.richTextBoxText.Focus();
}

```

双击最后一个格式化按钮 Center, 添加下列代码:

```

private void buttonCenter_Click(object sender, System.EventArgs e)
{
    if (this.richTextBoxText.SelectionAlignment == HorizontalAlignment.Center)
        this.richTextBoxText.SelectionAlignment = HorizontalAlignment.Left;
    else
        this.richTextBoxText.SelectionAlignment = HorizontalAlignment.Center;
    this.richTextBoxText.Focus();
}

```

这里必须检查另一个属性 SelectionAlignment, 看看选中的文本是否已经居中对齐, 因为我们希望按钮像一个切换按钮那样运作。如果文本已居中, 就使它左对齐, 否则就使它居中。HorizontalAlignment 是一个枚举, 其值可以是 Left、Right、Center、Justify 和 NotSet。在本例中, 只检查一下是否设置了 Center, 如果已经设置了, 就把对齐方式设置为 Left。如果不是, 就设置为 Center。

文本编辑器能进行的最后一个格式化操作是设置文本的大小。为文本框 Size 添加两个事件处理程序, 一个处理程序控制输入, 另一个处理程序检测用户输入完一个值的时间。

在 Properties 窗口的 Events 列表中找到并双击 textBoxSize 控件的 KeyPress 和 Validated 事件, 给处理程序添加代码。

与前面示例使用的 Validating 事件不同, Validated 事件在进行完验证后引发。这两个事件处理程序都使用一个帮助方法 ApplyTextSize, 该方法带有一个字符串参数, 表示文本的大小:

```

private void textBoxSize_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((e.KeyChar < 48 || e.KeyChar > 57) &&
        e.KeyChar != 8 && e.KeyChar != 13)
        e.Handled = true;
    else if (e.KeyChar == 13)
    {
        TextBox txt = (TextBox)sender;

        if (txt.Text.Length > 0)
            ApplyTextSize(txt.Text);
        e.Handled = true;
        this.richTextBoxText.Focus();
    }
}

private void textBoxSize_Validated(object sender, CancelEventArgs e)
{
    ApplyTextSize(txt.Text);
    this.richTextBoxText.Focus();
}

```

```
private void ApplyTextSize(string textSize)
{
    float newSize = Convert.ToSingle(textSize);
    FontFamily currentFontFamily;
    Font newFont;

    currentFontFamily = this.richTextBoxText.SelectionFont.FontFamily;
    newFont = new Font(currentFontFamily, newSize);

    this.richTextBoxText.SelectionFont = newFont;
}
```

KeyPress 事件只允许用户输入一个整数，并在用户按下回车键时，调用 ApplyTextSize。我们感兴趣的是帮助方法 ApplyTextSize()。它首先把文本的大小从字符串转换为浮点数。如前所述，我们只允许用户输入整数，但在创建新字体时，需要使用浮点数，所以把它转换为正确的类型。

之后，获取字体所属的字体系列，从该系列中创建一个带有新字号的新字体。最后，把选中文本的字体设置为新字体。

这就是我们所能进行的所有格式化操作，有一些操作可以由 RichTextBox 本身处理。如果现在尝试运行这个示例，就可以把文本设置为黑体、斜体和下划线，还可以使文本居中对齐。这就是我们期望的操作，但还有一些比较有趣的操作。试着在文本中键入一个网址，例如 <http://www.wrox.com>，该文本就被控件识别为一个 Internet 地址，加上下划线，当把鼠标指针移到该文本上时，鼠标指针就会变成手的形状。再添加一些代码，就可以在单击该文本时打开一个网页，为此，需要处理用户单击链接时引发的事件：LinkClicked。

在 Properties 窗口的 Events 列表中找到 LinkClicked 事件，双击它，在事件处理程序中添加代码。我们以前没有见过这个事件处理程序。它用于提供被单击的链接的文本，处理程序非常简单，如下所示：

```
private void richTextBoxText_LinkedClick(object sender,
                                         System.Windows.Forms.LinkClickedEventArgs e)
{
    System.Diagnostics.Process.Start(e.LinkText);
}
```

这段代码打开了默认的浏览器(如果浏览器没有打开)，并导航到该链接指向的站点。

应用程序的编辑部分就完成了。剩下的是加载和保存控件的内容。这里使用一个固定的文件。双击 Load 按钮，添加下面的代码：

```
private void buttonLoad_Click(object sender, EventArgs e)
{
    try
    {
        richTextBoxText.LoadFile("Test.rtf");
    }
    catch (System.IO.FileNotFoundException)
    {
        MessageBox.Show("No file to load yet");
    }
}
```

这就完成了，不需要做其他工作。因为我们处理的是文件，所以总是有可能遇到异常，必须处



理这些异常。在 Load 方法中，处理了因文件不存在而抛出的异常。保存文件也很简单，双击 Save 按钮，添加下面的代码：

```
private void buttonSave_Click(object sender, EventArgs e)
{
    try
    {
        richTextBoxText.SaveFile("Test.rtf");
    }
    catch (System.Exception err)
    {
        MessageBox.Show(err.Message);
    }
}
```

现在运行示例，格式化一些文本，再单击 Save 按钮。清空文本框，单击 Load 按钮，刚才保存的文本就会再次显示出来。  
运行它时，应得到如图 15-10 所示的结果。

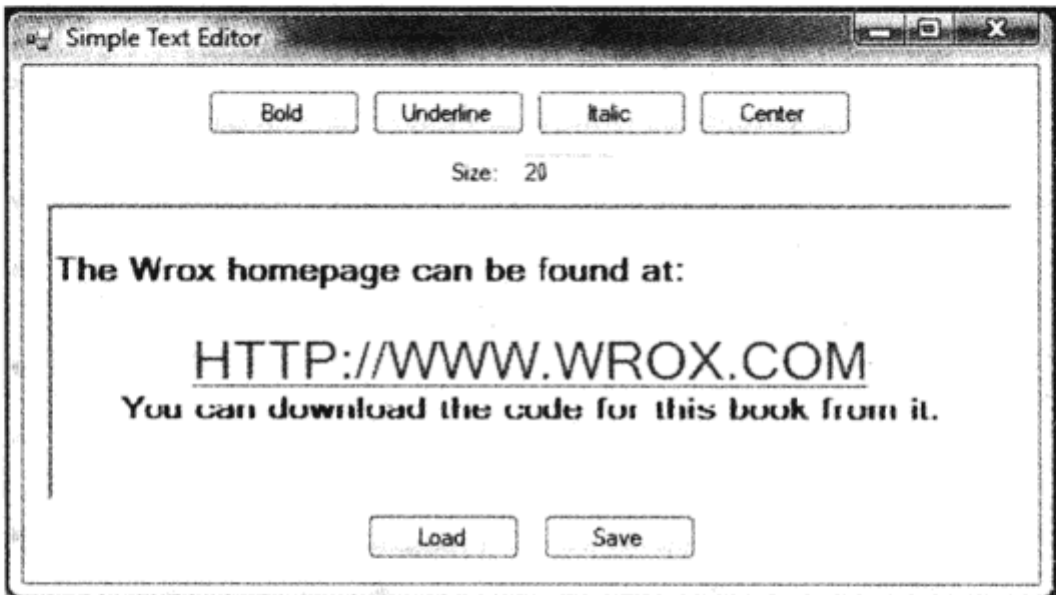


图 15-10

## 15.7 ListBox 和 CheckedListBox 控件

列表框用于显示一组字符串，可以一次从中选择一个或多个选项。与复选框和单选按钮一样，列表框也提供了要求用户选择一个或多个选项的方式。在设计期间，如果不知道用户要选择的数值个数，就应使用列表框(例如同事列表)。即使在设计期间知道所有可能的值，但列表中的值非常多，也应考虑使用列表框。

ListBox 类派生于 ListControl 类。后者提供了 .NET Framework 内置列表类型控件的基本功能。

另一类列表框称为 CheckedListBox，派生于 ListBox 类。它提供的列表类似于 ListBox，但除了文本字符串以外，每个列表选项还附带一个复选标记。

### 15.7.1 ListBox 控件的属性

除非明确声明，表 15-14 中列出的所有属性都可用于 ListBox 类和 CheckedListBox 类。

表 15-14

属 性	说 明
SelectedIndex	这个值表示列表框中选中项的基于 0 的索引。如果列表框可以一次选择多个选项，这个属性就包含选中列表中第一个选项的索引
ColumnWidth	在包含多个列的列表框中，这个属性指定列宽
Items	Items 集合包含列表框中的所有选项，使用这个集合的属性可以增加和删除选项
MultiColumn	列表框可以有多个列。使用这个属性可以获取是否采用多列形式的信息，也可以设置是否采用多列形式
SelectedIndices	这个属性是一个集合，包含列表框中选中项的所有基于 0 的索引
SelectedItem	在只能选择一个选项的列表框中，这个属性包含选中的选项。在可以选择多个选项的列表框中，这个属性包含选中项中的第一项
SelectedItems	这个属性是一个集合，包含当前选中的所有选项
SelectionMode	在列表框中，可以使用 ListSelectionMode 枚举中的 4 种选择模式： <ul style="list-style-type: none"><li>• None: 不能选择任何选项</li><li>• One: 一次只能选择一个选项</li><li>• MultiSimple: 可以选择多个选项。使用这个模式，在单击列表中的一项时，该项就会被选中，即使单击另一项，该项也仍保持选中状态，除非再次单击它</li><li>• MultiExtended: 可以选择多个选项，用户还可以使用 Ctrl、Shift 和箭头键进行选择。它与 MultiSimple 不同，如果先单击一项，然后单击另一项，则只选中第二个单击的项</li></ul>
Sorted	把这个属性设置为 true，会使列表框对它包含的选项按字母顺序排序
Text	许多控件都有 Text 属性。但这个 Text 属性与其他控件的 Text 属性大不相同。如果设置列表框控件的 Text 属性，它将搜索匹配该文本的选项，并选择该选项。如果获取 Text 属性，返回的值是列表中第一个选中的选项。如果 SelectionMode 是 None，就不能使用这个属性
CheckedIndices	(只用于 CheckedListBox)这个属性是一个集合，包含 CheckedListBox 中状态是 checked 或 indeterminate 的所有选项的索引
CheckedItems	(只用于 CheckedListBox)这是一个集合，包含 CheckedListBox 中状态是 Checked 或 Indeterminate 的所有选项
CheckOnClick	(只用于 CheckedListBox)如果这个属性是 true，则选项就会在用户单击它时改变它的状态
ThreeDCheckBoxes	(只用于 CheckedListBox)设置这个属性，就可以选择平面或正常的 CheckBoxes

15.7.2 ListBox 控件的方法

为了高效地操作列表框，读者应了解它可以调用的一些方法。表 15-15 列出了最常用的方法。除非特别声明，否则这些方法均属于 ListBox 和 CheckedListBox 类。

表 15-15

方 法	说 明
ClearSelected()	清除列表框中的所有选中项
FindString()	查找列表框中第一个以指定字符串开头的字符串，例如 FindString("a")就是查找列表框中第一个以 a 开头的字符串
FindStringExact()	与 FindString 类似，但必须匹配整个字符串
GetSelected()	返回一个表示是否选择一个选项的值
SetSelected()	设置或清除选项的选中状态
ToString()	返回当前选中的选项
GetItemChecked()	(只用于 CheckedListBox)返回一个表示选项是否被选中的值
GetItemCheckState()	(只用于 CheckedListBox)返回一个表示选项的选中状态的值
SetItemChecked()	(只用于 CheckedListBox)设置指定为选中状态的选项
SetItemCheckState()	(只用于 CheckedListBox)设置选项的选中状态

15.7.3 ListBox 控件的事件

通常，在处理 ListBox 和 CheckedListBox 时，使用的事件都与用户选中的选项有关，如表 15-16 所示。

表 15-16

事 件	说 明
ItemCheck	(只用于 CheckedListBox)在列表框中一个选项的选中状态改变时引发该事件
SelectedIndexChanged	在选中选项的索引改变时引发该事件

下面用 ListBox 和 CheckedListBox 创建一个小示例。用户可以查看 CheckedListBox 中的选项，然后单击一个按钮，把选中的选项移动到一般的 ListBox 中。

试一试：使用 ListBox 控件

创建如下所示的对话框：

- (1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 Lists。
- (2) 在窗体上添加一个 ListBox、一个 CheckedListBox 和一个按钮，改变其名称，如图 15-11 所示。
- (3) 把按钮的 Text 属性改成 Move。
- (4) 把 CheckedListBox 的属性 CheckOnClick 改成 true。
- (5) 单击省略号(...), 打开 CheckedListBox 控件的 Items 编辑器。再输入 One、Two、Three、Four、Five、Six、Seven、Eight 和 Nine，一项占一行，然后单击 OK。

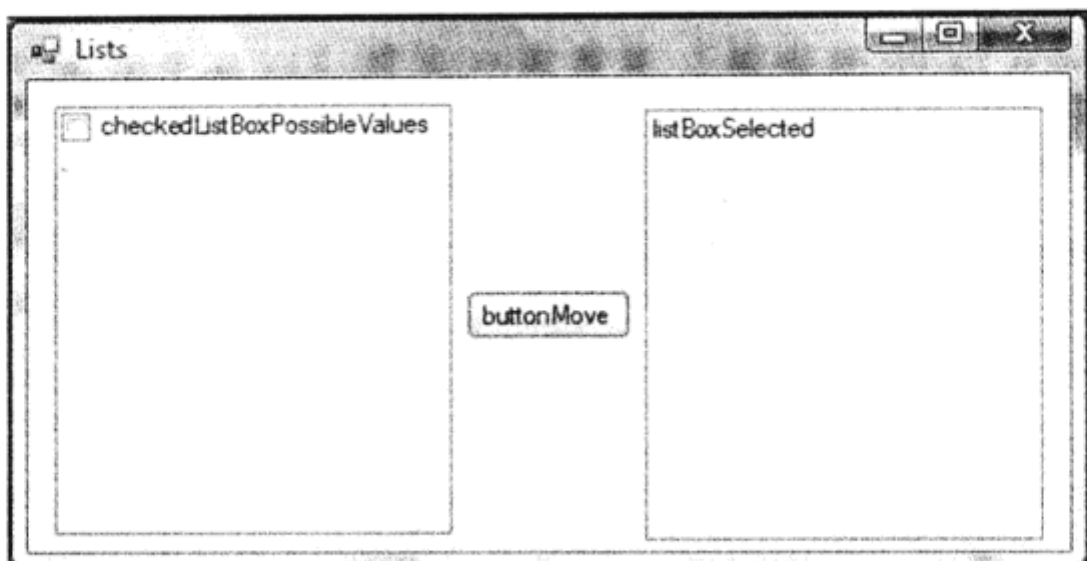


图 15-11

(6) 在 `CheckedListBox` 中再插入一项，但在代码中插入：



可从  
wrox.com  
下载源代码

```
public Form1()
{
    InitializeComponent();

    checkedListBoxPossibleValues.Items.Add("Ten");
}
```

代码段 Chapter15\Lists\Form1.cs

现在准备添加事件处理程序。可以添加一些代码，把 `checkedListBox` 中的项移动到正常的列表框中。当用户单击 `Move` 按钮时，要查找被选中的选项，再把它们复制到右边的列表框中。

(7) 双击该按钮，输入下面的代码：

```
private void buttonMove_Click(object sender, EventArgs e)
{
    if (checkedListBoxPossibleValues.CheckedItems.Count > 0)
    {
        listBoxSelected.Items.Clear();
        foreach (string item in checkedListBoxPossibleValues.CheckedItems)
        {
            listBoxSelected.Items.Add(item.ToString());
        }
        for (int i = 0; i < checkedListBoxPossibleValues.Items.Count; i++)
            checkedListBoxPossibleValues.SetItemChecked(i, false);
    }
}
```

#### 示例的说明

首先查看一下 `CheckedItems` 集合的 `Count` 属性。如果集合中有选中的选项，该属性就会大于 0。接着清除 `listBoxSelected` 列表框中的所有选项，循环 `CheckedItems` 集合，把每个选项添加到 `listBoxSelected` 列表框中。最后，删除 `CheckedListBox` 中的所有选中标记。

如果现在运行应用程序，就可以在左边选择一些项，再单击 `Move` 按钮，把它们移动到右边。这就结束了列表框的演示，下面看一个比较新的控件 `ListView`。

## 15.8 ListView 控件

图 15-12 显示了 Windows 中最常用的 ListView 控件。Windows 为显示文件和文件夹提供了许多其他方式，ListView 控件就包含其中一些选项，例如，显示大图标和详细视图等。

Name	Size	Type	Date Modified
Images		File Folder	01-05-2004 02:54
Magic		File Folder	20-02-2004 23:43
Programming		File Folder	09-07-2004 21:35
RECYCLER		File Folder	08-03-2004 22:46
System Volume Information		File Folder	11-02-2004 20:07
Temp		File Folder	24-05-2004 21:47
Web site		File Folder	20-03-2004 00:52

图 15-12

列表视图通常用于显示数据，用户可以对这些数据和显示方式进行某些控制。还可以把包含在控件中的数据显示为列和行(像网格那样)，或者显示为一列，或者显示为图标表示。最常用的列表视图就是图 15-12 中用于导航计算机中文件夹的视图。

ListView 控件是本章中最复杂的一个控件，它包括了超出本书范围的内容，这里仅编写一个示例，使用 ListView 控件中最重要的功能，为用户打下坚实的基础，将全面介绍可以使用的许多属性、事件和方法。本章还将讨论 ImageList 控件，它用于存储在 ListView 控件中使用的图像。

### 15.8.1 ListView 控件的属性

ListView 的属性如表 15-17 所示。

表 15-17

属 性	说 明
Activation	使用这个属性，可以控制用户在列表视图中激活选项的方式。可能的值如下： <ul style="list-style-type: none"><li>Standard: 这个设置是用户为自己的计算机选择的值</li><li>OneClick: 单击一个选项，激活它</li><li>TwoClick: 双击一个选项，激活它</li></ul>
Alignment	这个属性可以控制列表视图中选项的对齐方式。有 4 个可能的值： <ul style="list-style-type: none"><li>Default: 如果用户拖放一个选项，它将仍位于拖动前的位置</li><li>Left: 选项与 ListView 控件的左边界对齐</li><li>Top: 选项与 ListView 控件的顶边界对齐</li><li>SnapToGrid: ListView 控件包含一个不可见的网格，选项都放在该网格中</li></ul>
AllowColumn Reorder	如果把这个属性设置为 true，就允许用户改变列表视图中列的顺序。如果这么做，就应确保即使改变了列的属性顺序，填充列表视图的例程也能正确插入选项

(续表)

属 性	说 明
AutoArrange	如果把这个属性设置为 true，选项会自动根据 Alignment 属性排序。如果用户把一个选项拖放到列表视图的中央，且 Alignment 是 Left，则选项会自动左对齐。只有在 View 属性是 LargeIcon 或 SmallIcon 时，这个属性才有意义
CheckBoxes	如果把这个属性设置为 true，列表视图中的每个选项会在其左边显示一个复选框。只有在 View 属性是 Details 或 List 时，这个属性才有意义
CheckedIndices CheckedItems	利用这两个属性分别可以访问索引和选项的集合，该集合包含列表中被选中的选项
Columns	列表视图可以包含列。通过这个属性可以访问列集合，通过该集合，可以增加或删除列
FocusedItem	这个属性包含列表视图中有焦点的选项。如果没有选择任何选项，该属性就为 null
FullRowSelect	这个属性为 true 时，单击一个选项，该选项所在的整行文本都会突出显示。如果该属性为 false，则只有选项本身会突出显示
GridLines	把这个属性设置为 true，则列表视图会在行和列之间绘制网格线。只有 View 属性为 Details 时，这个属性才有意义
HeaderStyle	可以控制列标题的显示方式，有 3 种样式： <ul style="list-style-type: none"><li>• Clickable: 列标题显示为一个按钮</li><li>• NonClickable: 列标题不响应鼠标单击</li><li>• None: 不显示列标题</li></ul>
HoverSelection	这个属性设置为 true 时，用户可以把鼠标指针放在列表视图的一个选项上，以选择它
Items	列表视图中的选项集合
LabelEdit	这个属性设置为 true 时，用户可以在 Details 视图下编辑第一列的内容
LabelWrap	如果这个属性是 true 时，标签就会自动换行，以便显示所有文本
LargeImageList	这个属性包含 ImageList，而 ImageList 包含大图像。这些图像可以在 View 属性为 LargeIcon 时使用
MultiSelect	这个属性设置为 true 时，用户可以选择多个选项
Scrollable	这个属性设置为 true 时，就显示滚动条
SelectedIndices SelectedItems	这两个属性分别包含选中索引和选项的集合
SmallImageList	当 View 属性为 SmallIcon 时，这个属性包含了 ImageList，其中 ImageList 包含了要使用的图像

(续表)

属 性	说 明
Sorting	可以让列表视图对它包含的选项排序，有 3 种可能的模式： <ul style="list-style-type: none"><li>• Ascending</li><li>• Descending</li><li>• None</li></ul>
StateImageList	ImageList 包含图像的蒙板，这些图像蒙板可用作 LargeImageList 和 SmallImageList 图像的覆盖图，表示定制的状态
TopItem	返回列表视图顶部的选项
View	列表视图可以用 4 种不同的基本模式显示其选项： <ul style="list-style-type: none"><li>• LargeIcon: 所有选项都在其旁边显示一个大图标(32x32)和一个标签</li><li>• SmallIcon: 所有选项都在其旁边显示一个小图标(16x16)和一个标签</li><li>• List: 只显示一列。该列可以包含一个图标和一个标签</li><li>• Details: 可以显示任意数量的列。只有第一列可以包含图标</li><li>• Tile: (只用于 Windows XP 和较新的 Windows 平台)显示一个大图标和一个标签，在图标的右边显示子项信息</li></ul>

15.8.2 ListView 控件的方法

对于像列表视图这样复杂的控件来说，专用的方法非常少。表 15-18 列出了这些方法。

表 15-18

方 法	说 明
BeginUpdate()	调用这个方法，将告诉列表视图停止更新，直到调用 EndUpdate()为止。当一次插入多个选项时使用这个方法很有用，因为它会禁止视图闪烁，大大提高速度
Clear()	彻底清除列表视图，删除所有的选项和列
EndUpdate()	在调用 BeginUpdate 之后调用这个方法。在调用这个方法时，列表视图会显示其所有选项
EnsureVisible()	在调用这个方法时，列表视图会滚动，以显示指定索引的选项
GetItemAt()	返回列表视图中位于 x,y 位置的选项

15.8.3 ListView 控件的事件

表 15-19 列出了要处理的 ListView 控件的事件。



表 15-19

事 件	说 明
AfterLabelEdit	在编辑了标签后，引发该事件
BeforeLabelEdit	在用户开始编辑标签前，引发该事件
ColumnClick	在单击一个列时，引发该事件
ItemActivate	在激活一个选项时，引发该事件

15.8.4 ListViewItem

列表视图中的选项总是 ListViewItem 类的一个实例。ListViewItem 包含要显示的信息，如文本和图标索引。ListViewItems 对象有一个 SubItems 属性，其中包含另一个类 ListViewSubItem 的实例。如果 ListView 控件处于 Details 或 Tile 模式下，这些子选项就会显示出来。每个子选项表示列表视图中的一列。子选项和主选项之间的区别是，子选项不能显示图标。

通过 Items 集合把 ListViewItems 添加到 ListView 中，通过 ListViewItem 上的 SubItems 集合把 ListViewSubItems 添加到 ListViewItem 中。

15.8.5 ColumnHeader

要使列表视图显示列标题，需要把类 ColumnHeader 的实例添加到 ListView 的 Columns 集合中。当 ListView 控件处于 Details 模式下时，ColumnHeaders 为要显示的列提供一个标题。

15.8.6 ImageList 控件

ImageList 控件提供了一个集合，可以用于存储在窗体的其他控件中使用的图像。可以在图像列表中存储任意大小的图像，但在每个控件中，每个图像的大小必须相同。对于 ListView，则需要两个 ImageList 控件，才能显示大图像和小图像。

ImageList 是本章介绍的第一个不在运行期间显示自身的控件。在把它拖放到正在开发的窗体上时，它并不是放在窗体上，而是放在它的下面，其中包含所有的这类组件。这个功能可以防止不是用户界面一部分的控件把窗体设计器弄乱。这个控件的处理方式与其他控件相同，但不能把它移动到窗体上。

可以在设计和运行期间给 ImageList 添加图像。如果知道在设计期间需要显示哪些图像，就可以单击 Images 属性右边的按钮，添加这些图像。这会打开一个对话框，在该对话框中，可以浏览要插入的图像。如果选择在运行期间添加图像，就可以通过 Images 集合添加它们。

学习使用 ListView 控件及其相关的图像列表的最好方式是利用一个示例。下面的示例将创建一个对话框，其中有一个 ListView 和两个 ImageList。ListView 显示硬盘上的文件和文件夹。为简单起见，我们不提取文件和文件夹中的正确图标，而使用文件夹的标准文件夹图标和文件的文本图标。

双击文件夹，就可以浏览文件夹树，后退按钮可以在文件夹树中向上移动。5 个单选按钮用于在运行期间改变列表视图的模式。如果双击了一个文件，就可以执行它。

试一试：使用 ListView 控件

与往常一样，首先创建用户界面：

- (1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新 Windows 应用程序 ListView。
- (2) 在窗体上添加一个列表视图、一个按钮、一个标签和一个组框。然后在组框中添加 5 个单选按钮，此时窗体应如图 15-13 所示。要设置标签控件的宽度，应将其 AutoSize 属性设置为 False。把标签控件设置得与列表视图一样宽。
- (3) 如图 15-13 所示命名控件。ListView 不在图中显示其名称，这里添加了一个选项，显示其名称。读者不需要这么做。

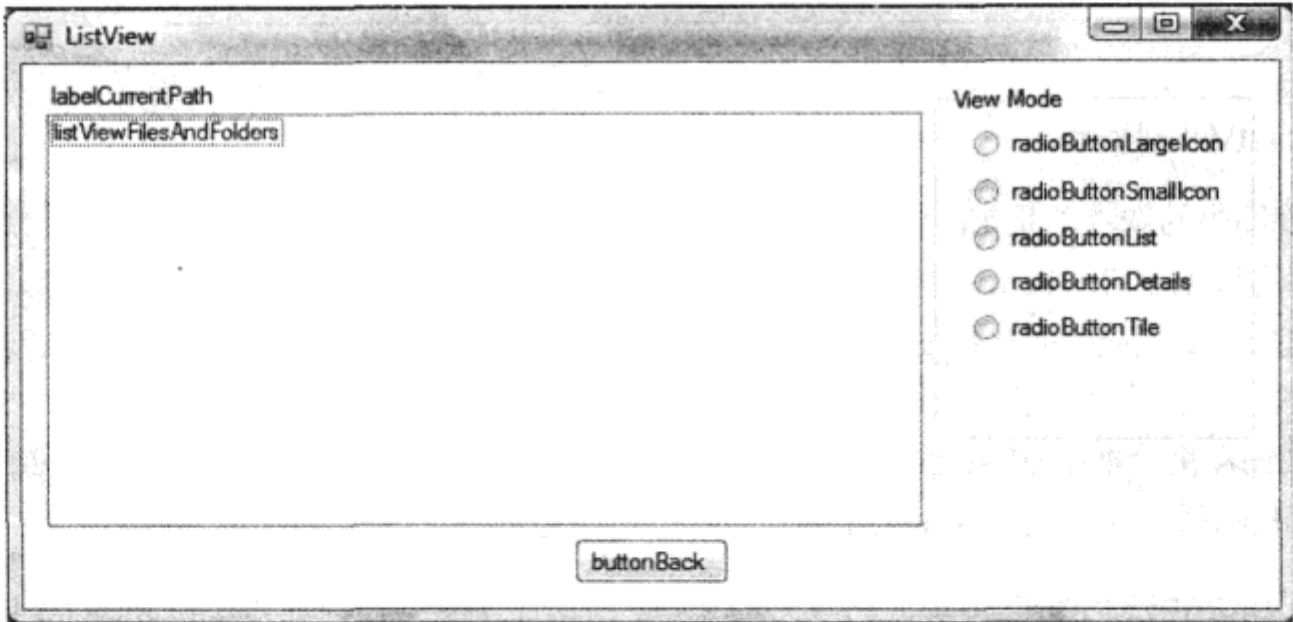


图 15-13

- (4) 把单选按钮的 Text 属性值改为其名称(但控件名称除外)，把窗体的 Text 属性设置为 ListView。
- (5) 清空标签的 Text 属性。
- (6) 在工具箱中双击 ImageList 控件的图标，在窗体中添加两个图像列表。ImageList 控件在工具箱的 Components 区域中。把它们重新命名为 imageListSmall 和 imageListLarge。
- (7) 把图像列表 imageListLarge 的 Size 属性值改为 32, 32。
- (8) 单击 imageListLarge 图像列表的 Images 属性右边的按钮，打开一个对话框，从中可以浏览要插入的图像。
- (9) 单击 Add，浏览本章代码的 ListView 文件夹。这些文件是 Folder 32x32.ico 和 Text 32x32.ico。
- (10) 确保文件夹图标位于列表顶部。
- (11) 对另一个图像列表 imageListSmall 重复第(8)、(9)步，选择 16×16 版本的图标。
- (12) 把单选按钮 radioButtonDetails 的 Checked 属性设置为 true。
- (13) 设置列表视图的属性，如表 15-20 所示。

表 15-20

属 性	值
LargeImageList	imageListLarge
SmallImageList	imageListSmall
View	Details

## 添加事件处理程序

这是我们的用户界面，下面可以添加代码了。首先，需要一个字段，以包含前面浏览的文件夹，在单击后退按钮时，就可以返回这些文件夹。我们将存储文件夹的绝对路径，所以选择使用 `StringCollection`：

```
partial class Form1 : Form
{
    private System.Collections.Specialized.StringCollection folderCol;
```

在窗体设计器中没有创建任何列标题，现在要使用 `CreateHeadersAndFillListView()` 方法在代码中创建：



```
private void CreateHeadersAndFillListView()
{
    ColumnHeader colHead;

    colHead = new ColumnHeader();
    colHead.Text = "Filename";
    listViewFilesAndFolders.Columns.Add(colHead); // Insert the header

    colHead = new ColumnHeader();
    colHead.Text = "Size";
    listViewFilesAndFolders.Columns.Add(colHead); // Insert the header

    colHead = new ColumnHeader();
    colHead.Text = "Last accessed";
    listViewFilesAndFolders.Columns.Add(colHead); // Insert the header
}
```

代码段 Chapter15\ListView\Form1.cs.

先声明一个变量 `colHead`，用于创建 3 个列标题。这 3 个标题都是用 `new` 关键字创建的，在把它添加到 `ListView` 的 `Columns` 集合中之前，将文本赋给它。

在第一次显示窗体时，进行的最后一个初始化工作是用硬盘上的文件和文件夹填充列表视图。这通过另一个方法来完成：



```
private void PaintListView(string root)
{
    try
    {
        ListViewItem lvi;
        ListViewItem.ListViewSubItem lvsi;

        if (string.IsNullOrEmpty(root))
            return;

        DirectoryInfo dir = new DirectoryInfo(root);
        DirectoryInfo[] dirs = dir.GetDirectories();
        FileInfo[] files = dir.GetFiles();
```

```

listViewFilesAndFolders.Items.Clear();
labelCurrentPath.Text = root;
listViewFilesAndFolders.BeginUpdate();

foreach (DirectoryInfo di in dirs)
{
    lvi = new ListViewItem();
    lvi.Text = di.Name;
    lvi.ImageIndex = 0;
    lvi.Tag = di.FullName;

    lvsi = new ListViewItem.ListViewSubItem();
    lvsi.Text = "";
    lvi.SubItems.Add(lvsi);

    lvsi = new ListViewItem.ListViewSubItem();
    lvsi.Text = di.LastAccessTime.ToString();
    lvi.SubItems.Add(lvsi);
    listViewFilesAndFolders.Items.Add(lvi);
}
foreach (FileInfo fi in files)
{
    lvi = new ListViewItem();
    lvi.Text = fi.Name;
    lvi.ImageIndex = 1;
    lvi.Tag = fi.FullName;

    lvsi = new ListViewItem.ListViewSubItem();
    lvsi.Text = fi.Length.ToString();
    lvi.SubItems.Add(lvsi);

    lvsi = new ListViewItem.ListViewSubItem();
    lvsi.Text = fi.LastAccessTime.ToString();
    lvi.SubItems.Add(lvsi);
    listViewFilesAndFolders.Items.Add(lvi);
}

listViewFilesAndFolders.EndUpdate();
}
catch (System.Exception err)
{
    MessageBox.Show("Error: " + err.Message);
}
}

```

---

代码段 Chapter15\ListView\Form1.cs.

---

### 示例的说明

在第一个 foreach 块中, 对 ListView 控件调用了 BeginUpdate()。ListView 控件上的 BeginUpdate() 方法告诉 ListView 控件, 停止更新其可见区域, 直到调用了 EndUpdate() 为止。如果没有调用这个方法, 列表视图的填充就会进行得更加缓慢, 列表可能在填充选项时闪烁。在第二个 foreach 块的后面调用了 EndUpdate(), 就可以使 ListView 控件显示出填充到它里面的内容。

这两个 foreach 块包含了我们感兴趣的代码。首先创建 ListViewItem 的一个新实例，再把 Text 属性设置为要插入的文件名或文件夹名。ListViewItem 的 ImageIndex 表示其中一个 ImageList 中的选项索引。所以两个 ImageList 中的图标有相同的索引是非常重要的。使用 Tag 属性保存文件夹和文件的完全限定路径，在用户双击选项时，将使用该路径。

然后创建两个子选项，将要显示的文本赋给这两个子选项，再把它们添加到 ListViewItem 的 SubItems 集合中。

最后，把 ListViewItem 添加到 ListView 的 Items 集合中。ListView 非常聪明，知道如果视图模式不是 Details，就应忽略子选项。所以，现在无论视图模式是什么，都可以增加子选项。

注意代码的某些方面没有讨论，即实际获取文件信息的代码行：

```
// Get information about the root folder.
DirectoryInfo dir = new DirectoryInfo(root);
// Retrieve the files and folders from the root folder.
DirectoryInfo[] dirs = dir.GetDirectories();
FileInfo[] files = dir.GetFiles();
```

这些代码使用 System.IO 名称空间中的类访问文件，所以需要在代码顶部的 using 区域添加如下代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

第 24 章将详细介绍文件访问和 System.IO 名称空间，但现在应明白，DirectoryInfo 对象上的 GetDirectories() 方法返回一个对象集合，它们表示所查看的目录下的文件夹，GetFiles() 方法返回一个对象集合，它们表示当前目录下的文件。可以迭代这些集合，如上面的代码所示，使用对象的 Name 属性返回相关目录或文件的名称，创建一个 ListViewItem 来保存这个字符串。

剩下的就是列表视图应显示根文件夹，为此，在窗体的构造函数中调用两个函数。同时用根文件夹实例化 folderCol StringCollection 字符串集合：

```
InitializeComponent();

folderCol = new System.Collections.Specialized.StringCollection();
CreateHeadersAndFillListView();
PaintListView(@"C:\");
folderCol.Add(@"C:\");
```

为了允许用户通过双击 ListView 中的选项来浏览文件夹，需要订阅 ItemActivate 事件。在设计器中选择 ListView，在 Properties 窗口的 Events 列表中双击 ItemActivate 事件。

对应的事件处理程序如下所示：

```
private void listViewFilesAndFolders_ItemActivate(object sender, EventArgs e)
{
    System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
    string filename = lw.SelectedItems[0].Tag.ToString();
```

```

        if (lw.SelectedItems[0].ImageIndex != 0)
        {
            try
            {
                System.Diagnostics.Process.Start(filename);
            }
            catch {return;}
        }
        else
        {
            PaintListView(filename);
            folderCol.Add(filename);
        }
    }
}

```

选中项的 Tag 包含被双击的文件或文件夹的完全限定路径。索引为 0 的图像是一个文件夹，所以查看索引就可以确定哪个选项是文件，哪个选项是文件夹。如果选项是一个文件，就试着加载它。如果选项是一个文件夹，就通过新文件夹调用 PaintListView()，再把新文件夹添加到 folderCol 集合中。

在讨论单选按钮前，先给 Back 按钮添加 Click 事件，提供完整的浏览功能。双击该按钮，为事件处理程序添加如下代码：

```

private void buttonBack_Click(object sender, EventArgs e)
{
    if (folderCol.Count > 1)
    {
        PaintListView(folderCol[folderCol.Count-2].ToString());
        folderCol.RemoveAt(folderCol.Count-1);
    }
    else
        PaintListView(folderCol[0].ToString());
}

```

如果 folderCol 集合中有多个选项，我们就不在浏览器的根文件夹下，对该路径调用 PaintListView()，进入上面的文件夹。folderCol 集合中的最后一个选项是当前文件夹，这就是需要第二次提取最后一个选项的原因。然后删除集合中的最后一个选项，使前面一个选项成为当前文件夹。如果该集合中只有一个选项，就只需对该选项调用 PaintListView()。

剩下的是修改列表视图的查看类型。双击每个单选按钮，添加如下代码：



可从  
wrox.com  
下载源代码

```

private void radioButtonLargeIcon_CheckedChanged(object sender, EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.listViewFilesAndFolders.View = View.LargeIcon;
}

private void radioButtonList_CheckedChanged(object sender, EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.listViewFilesAndFolders.View = View.SmallIcon;
}

```

```
}

private void radioButtonSmallIcon_CheckedChanged(object sender, EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.listViewFilesAndFolders.View = View.List;
}

private void radioButtonDetails_CheckedChanged(object sender, EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.listViewFilesAndFolders.View = View.Details;
}

private void radioButtonTile_CheckedChanged(object sender, EventArgs e)
{
    RadioButton rdb = (radioButton)sender;
    if (rdb.Checked)
        this.listViewFilesAndFolders.View = View.Tile;
}
```

代码段 Chapter15\ListView\Form1.cs.

检查单选按钮，看看是否已将其改为 Checked。如果是，就设置 ListView 的 View 属性。这就是 ListView 示例。运行它，将得到如图 15-14 所示的结果。

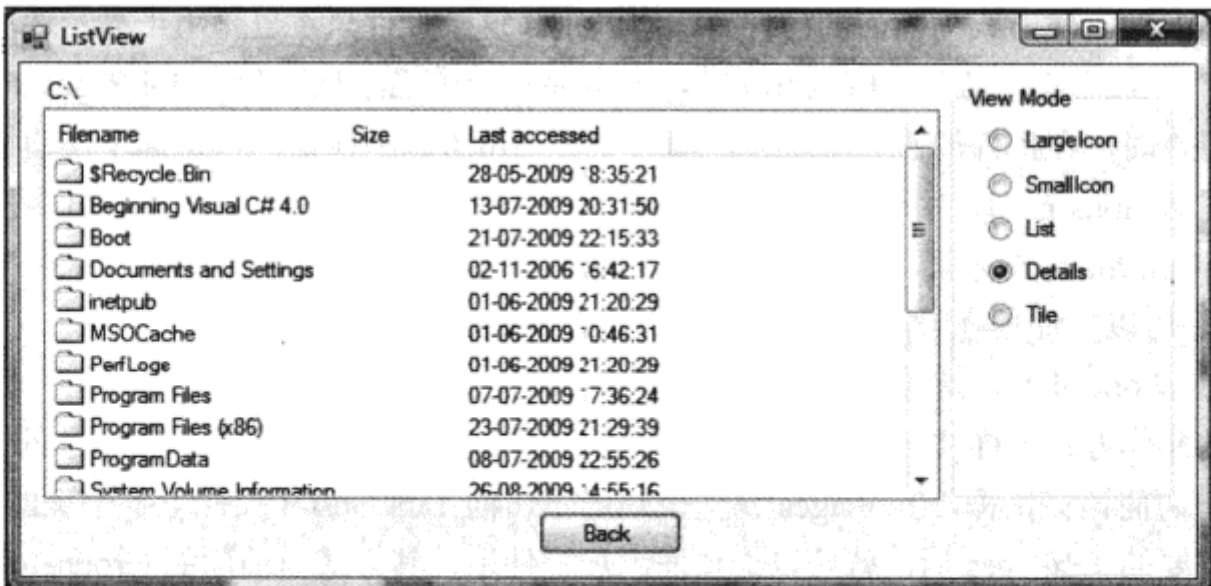


图 15-14

## 15.9 TabControl 控件

TabControl 提供了一种简单的方式，可以把对话框组织为合乎逻辑的部分，以便根据控件顶部的选项卡来访问。TabControl 包含 TabPages，TabPages 的工作方式与 GroupBox 控件非常类似，也是把控件组合在一起，但它们复杂。

TabControl 控件的使用是非常简单的。可以在控件的 tabPage 对象集合中添加任意数量的选项



卡，再把要显示的控件拖放到各个页面上。

15.9.1 TabControl 控件的属性

TabControl 的属性(如表 15-21 所示)一般用于控制 TabPage 对象容器的外观，特别是显示的选项卡的外观。

表 15-21

属 性	说 明
Alignment	控制选项卡在选项卡控件的什么位置显示。默认位置为控件的顶部
Appearance	控制选项卡的显示方式。选项卡可以显示为一般的按钮或带有平面样式
HotTrack	如果这个属性设置为 true，则当鼠标指针滑过控件上的选项卡时，其外观就会改变
Multiline	如果这个属性设置为 true，就可以有几行选项卡
RowCount	返回当前显示的选项卡行数
SelectedIndex	返回或设置选中选项卡的索引
SelectedTab	返回或设置选中的选项卡。注意这个属性在 TabPages 的实例上使用
TabCount	返回选项卡的总数
TabPage	这是控件中的 TabPage 对象集合。使用这个集合可以添加和删除 TabPage 对象

15.9.2 使用 TabControl 控件

TabControl 的工作方式与前面的控件有一些区别。这个控件只不过是用于显示选项卡的选项卡页面的容器。在工具箱中双击 TabControl 时，就会显示一个已添加了两个 TabPages 的控件。

选择该控件时，在控件的右上角就会出现一个带三角形的小按钮。单击这个按钮，就会打开一个小窗口，即 Actions 窗口，用于访问控件的所选属性和方法。Visual Studio 中的许多控件都有这个特性，但 TabControl 是本章第一个允许在 Actions 窗口中执行某些操作的控件。使用 TabControl 的 Actions 窗口，可以方便地在设计期间添加和删除 TabPages。

上面给 TabControl 添加选项卡页的过程可以让用户很快使用和运行该控件。另一方面，如果要改变选项卡的操作方式或样式，就应使用 TabPages 对话框；在选择 Properties 窗口中的 TabPages 时，可以通过按钮访问该对话框。TabPage 属性也是用于访问 TabControl 控件上各个页面的集合。

添加了需要的 TabPages 后，就可以给页面添加控件了，其方式与前面的 GroupBox 相同。下面创建一个示例，说明该控件的基本内容。

试一试：使用标签页

按照下面的步骤创建一个 Windows 应用程序，说明如何把控件放在选项卡控件的不同页面上：

- (1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 TabControl。
- (2) 把一个 TabControl 控件从工具箱拖放到窗体上。与 GroupBox 一样，TabControl 在工具箱的 Containers 选项卡中。
- (3) 找到 TabPages 属性，选择它后，单击它右边的按钮，打开 TabPage Collection Editor。

(4) 把选项卡页的 Text 属性分别改成 Tab One 和 Tab Two。单击 OK，关闭该对话框。

(5) 单击控件顶部的选项卡，选择要处理的选项卡。选择标有 Tab One 的选项卡。在控件上拖放按钮。确保把该按钮放在 TabControl 的框架中。如果把它放在框架的外部，则该按钮就会放在窗体上，而不是选项卡控件上。

(6) 将按钮的名称改为 buttonShowMessage，将其 Text 属性改为 Show Message。

(7) 单击 Text 属性为 Tab Two 的选项卡，把一个文本框控件拖放到 TabControl 上。为这个控件命名。

(8) 这两个选项卡应如图 15-15 和 15-16 所示。

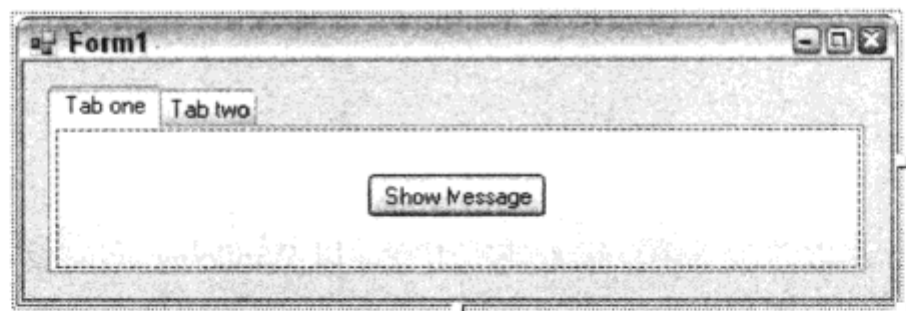


图 15-15

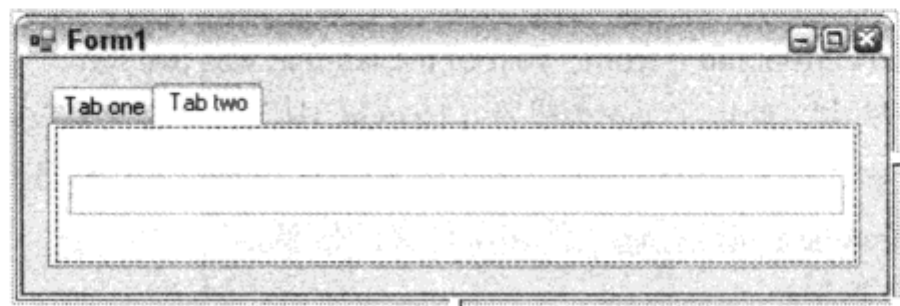


图 15-16

下面准备访问控件。如果运行代码，就会看到选项卡正确显示出来了。为了说明选项卡控件的用法，剩下要做的工作是添加一些代码，以便在用户单击一个选项卡上的 Show Message 按钮时，在另一个选项卡页中输入的文本将显示在消息框中。首先，双击第一个选项卡上的按钮，为 Click 事件添加一个处理程序，再添加下述代码：

```
private void buttonShowMessage_Click(object sender, EventArgs e)
{
    MessageBox.Show(textBoxMessage.Text);
}
```

#### 示例的说明

在选项卡页上访问一个控件，与访问窗体上的其他控件是一样的。获取文本框的 Text 属性，在消息框中显示它。

本章前面介绍过，在窗体中一次只能选择一个单选按钮(除非把它们放在组框中)。TabPage 与组框的工作方式完全相同，所以可以在不同的选项卡上放置多组单选按钮，而不需要使用组框。如 buttonShowMessage\_Click 方法所示，还可以访问位于其他选项卡上的控件。

要能处理选项卡控件，最后要注意的是如何确定当前显示的是哪个选项卡。这可以使用两个属性：SelectedTab 和 SelectedIndex，顾名思义，SelectedTab 返回 TabPage 对象，如果没有选择选项卡，

就返回 null。而 SelectedIndex 返回选项卡的索引，如果没有选择选项卡，就返回 - 1。练习题(2)将使用这些属性。

### 15.10 小结

本章介绍了创建 Windows 应用程序时最常用的一些控件，并讨论了如何使用它们创建简单而强大的用户界面。还论述了这些控件的属性和事件，列出了使用它们的示例，解释了如何为控件的特定事件添加处理程序。

第 16 章将讨论更复杂的控件，以及创建 Windows 窗体应用程序的特性。

### 15.11 练习

- (1) 在 Visual Studio 的以前版本中，很难使应用程序以 Windows 当前版本的样式显示其控件。本练习要在 Windows 窗体应用程序中，找到新的 Windows 窗体项目中启用各种可视样式的位置。试着启用和禁用该样式，看看这些操作对窗体上的控件有什么影响。
- (2) 修改 TabControl 示例，方式是添加几个选项卡页，在消息框中显示文本 You changed the current tab to <Text of the current tab> from <Text of the tab that was just left>。
- (3) 在 ListView 示例中，使用了 tag 属性在 ListView 中保存文件夹和文件的完全限定路径。修改这个操作，创建一个派生于 ListViewItem 的新类，使用这个新类的实例作为 ListView 中的项。在新类中使用 FullyQualifiedPath 属性存储文件和文件夹的信息。

附录 A 给出了练习答案。

### 15.12 本章要点

主 题	重 要 概 念
Label 控件	使用 Label 和 LinkLabel 控件给用户显示信息
Button 控件	使用 Button 控件和相应的 Click 事件，让用户告诉应用程序他们要进行什么操作
TextBox 控件	使用 TextBox 和 RichTextBox 控件，让用户输入纯文本或格式化文本
选项控件	区分 CheckBox 和 RadioButton 及其用法。还学习了如何把这两个控件组合到 GroupBox 控件中，以及这么做对控件有什么影响
ListBox 控件	使用 CheckedListBox 提供列表，用户可以单击复选框，从该列表中选择选项。还学习了如何使用更常见的 ListView 控件，提供与 CheckedListBox 控件类似的列表，但没有复选框
ListView 控件	使用 ListView 和 ImageList 控件提供一个列表，让用户以不同的方式查看
TabControl 控件	最后学习了如何使用 TabControl 把控件组合到同一个窗体的不同页面上，用户可以随意从这些页面上选择控件

# 第 16 章

## Windows 窗体的高级功能

### 本章内容:

- 使用 3 个常见控件创建外观多样的菜单、工具栏和状态栏
- 创建 MDI 应用程序
- 创建自己的控件

第 15 章介绍了 Windows 应用程序开发中一些最常用的控件。使用第 15 章介绍的控件，可以创建界面十分友好的对话框，但 Windows 应用程序的用户界面很少只包含一个对话框。这些应用程序使用单一文档界面(Single Document Interface, SDI)或者多文档界面(Multiple Document Interface, MDI)。这两种类型的应用程序通常会大量使用菜单和工具栏，本章就讨论它们。



.NET Framework 添加了 WPF(Windows Presentation Foundation)后，引入了几个新类型的 Windows 应用程序，详见第 25 章。

像第 15 章那样，本章在介绍控件时，首先介绍菜单控件，再介绍工具栏，说明如何把工具栏上的按钮与特定的菜单项链接起来，或把特定的菜单项与工具栏上的按钮链接起来。接着创建 SDI 和 MDI 应用程序，主要讨论 MDI 应用程序，因为 SDI 应用程序基本上是 MDI 应用程序的子集。

到目前为止，仅使用了 .NET Framework 提供的控件。这些控件非常强大，提供了许多功能，但有时候使用它们还是不够。所以需要创建定制控件，本章的最后介绍如何创建定制控件。

### 16.1 菜单和工具栏

有几个 Windows 应用程序不包含菜单或工具栏？这个数字可能接近于 0。在为 Windows 操作系统编写的应用程序中，菜单和工具栏可能是不可或缺的重要部分。为了帮助用户创建应用程序的菜单，Visual Studio 2010 提供了两个控件，使用它不必做太多的工作，就可以快速创建外观类似于 Visual

Studio 的菜单和工具栏。

### 16.1.1 两个实质一样的控件

下面要介绍的两个控件是 Visual Studio 2005 的新增控件，它们为开发人员和专业人士提供了许多强大的功能。仍旧在编写定制绘图处理程序和购买第三方组件的用户，使用这两个控件可以创建出工具栏和菜单具有专业化外观的应用程序。以前需要几个星期才能创建出的应用程序，现在变成在区区数秒内即可完成的简单任务。

我们要使用的控件包含在后缀为Strip的控件系列中，分别是ToolStrip、MenuStrip和StatusStrip。StatusStrip在本章后面介绍。从它们最纯粹的形式来看，ToolStrip和MenuStrip实际上是相同的控件，因为MenuStrip直接派生于ToolStrip。也就是说，ToolStrip可以做的工作，MenuStrip也能完成。显然，它们两个一起完成会更好。

### 16.1.2 使用MenuStrip控件

除了MenuStrip控件之外，还有许多控件可用于填充菜单。3个最常见的控件是ToolStripMenuItem、ToolStripDropDown和ToolStripSeparator。这些控件表示查看菜单或工具栏中某一项的特定方式。ToolStripMenuItem表示菜单中的一项，ToolStripDropDown表示单击一项，就会显示包含其他项目的一个列表，ToolStripSeparator表示菜单或工具栏中的水平或垂直分隔线。

在讨论完MenuStrip之后，还要讨论另一种菜单ContextMenuStrip。当用户右击一项时，关联菜单就会显示出来，它通常显示与该项相关的信息。

在下面的示例中，将创建本章的第一个示例。

**试一试：在5秒内创建具有专业外观的菜单**

第一个示例只是尝试一下，如果要创建外观非常标准的菜单，就应了解新控件的其他精彩方面。

(1) 在 C:\BegVCSharp\Chapter16 目录中创建一个新 Windows 应用程序，命名为 Professional Menus。

(2) 从工具箱中，把一个MenuStrip控件的实例拖放到设计界面上。

(3) 在对话框的顶部，单击MenuStrip右边的三角形，显示 Actions 窗口。

(4) 单击菜单右上角的三角形，再单击 Insert Standard Items 链接。

这就完成了。如果向下拖动 File 菜单，就会看到许多熟悉的选项，包括快捷键和图标。现在该菜单还没有什么功能——必须在其中填充内容。还可以编辑菜单，请继续往下看。

### 16.1.3 手工创建菜单

从工具箱中把MenuStrip控件拖放到设计界面上时，该控件会位于窗体和控件盘上，而且可以直接在窗体上编辑。要创建新的菜单项，只需把指针放在 Type Here 框上。

在突出显示的框中输入菜单的标题，在要用作该菜单项快捷键字符的字母前面加上一个宏字符(&)，在菜单项中，该字母显示为下划线形式，可以按下 Alt 键和该字母键来选择该菜单项。

注意，可以在一个菜单中用同一快捷键字符创建好几个菜单项，其规则是每个弹出菜单只能将该字符使用一次(例如，在 Files 弹出菜单中使用一次，在 View 菜单中使用一次等)。如果不小心把同一个快捷键字符赋予了同一个弹出菜单中的多个菜单项，则只有最靠近控件顶部的那个菜单项会



响应该字符。

选择一项，控件就会自动在当前项的下面和右边显示一些项。给这两个控件输入标题，就创建了与开始时选中的项相关的一个新项，这就是创建下拉菜单的方式。

要创建水平线，把菜单分成组，必须使用 `ToolStripSeparator` 控件，而不是 `ToolStripMenuItem` 控件，但不需要插入另一个控件。还可以键入一个短横线(-)，作为该项的标题。Visual Studio 会自动假定该项是一个分隔符，改变控件的类型。

下面的示例要创建一个菜单，但不使用 Visual Studio 生成其上的各项。

#### 试一试：从头创建菜单

在这个示例中，要从头创建 File 和 Help 菜单，Edit 和 Tools 菜单留给读者创建。

(1) 创建一个新的 Windows 应用程序项目，命名为 Manual Menus，保存在 C:\BegVCSharp\Chapter16 文件夹中。

(2) 把 `MenuStrip` 控件从工具栏拖放到设计界面上。

(3) 单击 `MenuStrip` 控件的 Type Here 文本区域，键入 `&File`，按下回车键。

(4) 在 File 项下面的文本区域键入如下内容：

- `&New`
- `&Open`
- `-`
- `&Save`
- `Save &As`
- `-`
- `&Print`
- `Print Preview`
- `-`
- `E&xit`

注意 Visual Studio 会自动把短线变成分隔各元素的线条。

(5) 单击 Files 右边的文本区域，键入 `&Help`。

(6) 在 Help 项下面的文本区域中键入以下内容：

- `Contents`
- `Index`
- `Search`
- `-`
- `About`

(7) 下面返回到 File 菜单，为菜单项设置快捷键。为此，选择要设置的菜单项，在 Properties 面板中找到 `ShortcutKeys` 属性，单击向下箭头，打开一个小窗口，在该窗口中可以设置与菜单项相关的键组合。由于这个菜单是一个标准菜单，因此应使用标准的键组合。如果要创建其他键组合，可以自由选择其他键组合。File 菜单中的 `ShortcutKeys` 属性设置如表 16-1 所示。

表 16-1

菜单项名称	属 性 和 值
&New	Ctrl + N
&Open	Ctrl + O
&Save	Ctrl + S
&Print	Ctrl + P

(8) 现在完成图像的处理。在 File 菜单中选择 New 菜单项，在属性面板的 Image 属性左边单击省略号(...)，打开 Select Resource 对话框。

在创建这些菜单时，最困难的地方是获取要显示的图像。在本例中，可以从 [www.wrox.com](http://www.wrox.com) 上下载本书的源代码来获得这些图像。但一般情况下应自己绘制图像，或者以其他方式获得。

(9) 由于目前项目中没有资源，所以 Entry 列表框是空的。单击 Import，这个示例的图像在本书源代码的 Chapter16\Manual Menus\Images 文件夹下。选择该文件夹下的所有文件，单击 Open。现在编辑的是 New 菜单项，所以在 Entry 列表中选择 New 图像，单击 OK 按钮。

(10) 对 Open、Save、Save As、Print 和 Print Preview 按钮的按钮重复第(9)步。

(11) 运行项目，注意可以单击 File 菜单，或者按下 Alt+F 组合键，来选择该菜单。Help 菜单可通过 Alt+H 组合键来选择。

16.1.4 ToolStripMenuItem 控件的其他属性

ToolStripMenuItem 有另外几个属性，在创建菜单时应了解这些属性。表 16-2 并不完整，如果需要完整的列表，可参阅 .NET Framework SDK 文档说明。

表 16-2

属 性	说 明
Checked	表示菜单是否被选中
CheckOnClick	这个属性是 true 时，如果菜单项文本左边的复选框没有打上标记，就打上标记，如果该复选框已打上了标记，就去除该标记，否则，该标记就被一个图像替代，使用 Checked 属性确定菜单项的状态
Enabled	把 Enabled 设置为 false，菜单项就会灰显，不能被选中
DropDownItems	这个属性返回一个项集合，用作与菜单项相关的下拉菜单

16.1.5 给菜单添加功能

下面就可以生成与 Visual Studio 的外观相同的菜单了。只需在单击菜单时执行某些操作即可。显然，这个操作取决于用户。在下面的示例中，将在上一个示例的基础上创建一个非常简单的示例。

为了响应用户做出的选择，就应为 ToolStripMenuItems 发送的两个事件之一提供处理程序。见表 16-3。



表 16-3

事件名称	说 明
Click	在用户单击菜单项时，引发该事件。大多数情况下这就是要响应的事件
CheckedChanged	当单击带 CheckOnClick 属性的菜单项时，引发这个事件

下面扩展上一个示例 Manual Menus，给对话框添加一个文本框，执行几个事件处理程序。在 File 和 Help 之间再添加一个菜单 Format。在下载代码中，这个项目的名称是 Extended Manual Menus。

试一试：处理菜单事件

- (1) 继续使用上一个示例创建的窗体，把一个 RichTextBox 拖放到设计界面上，把它的名称改为 richTextBoxText，其属性 Dock 设置为 Fill。
- (2) 选择 MenuStrip，在 Help 菜单项旁边的文本区域中输入 Format，按下回车键。
- (3) 选择 Format 菜单项，把它拖动到 Files 和 Help 之间。
- (4) 给 Format 菜单添加一个菜单项 Show Help Menu。
- (5) 把 Show Help Menu 菜单项的 CheckOnClick 属性设置为 true，将其 Checked 属性设置为 true。
- (6) 选择 showHelpMenuToolStripMenuItem，在属性面板的 Events 列表中双击 CheckedException 事件，为该事件添加处理程序。
- (7) 为事件处理程序添加如下代码：

```
private void showHelpMenuToolStripMenuItem_CheckedChanged(object sender,
EventArgs e)
{
    ToolStripMenuItem item = (ToolStripMenuItem)sender;
    helpToolStripMenuItem.Visible = item.Checked;
}
```

- (8) 双击 newToolStripMenuItem、saveToolStripMenuItem 和 openToolStripMenuItem，在设计视图中双击 ToolStripMenuItem，会把 Click 事件添加到代码中。输入下面的代码：

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
{
    richTextBoxText.Text = "";
}

private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    try
    {
        richTextBoxText.LoadFile(@"Example.rtf");
    }
    catch { }
}

private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
    try
    {
```

```
richTextBoxText.SaveFile("Example.rtf");
}
catch { }
}
```

(9) 运行应用程序，单击 Show Help Menu，Help 菜单就会消失或出现，这取决于 Checked 属性的状态。该程序应可以打开、保存和清除文本框中的文本。

示例的说明

先处理 ShowHelpMeunToolStripMenuItem\_CheckedChanged 事件。如果 Checked 属性是 true，这个事件的处理程序就应把 MenuItemHelp 的 Visible 属性设置为 true，否则就设置为 false。这会使该菜单项变成 Help 菜单的切换按钮。

最后，3 个 Click 事件处理程序分别清除 RichTextBox 中的文本，把其中的内容保存到预定义的文件中，打开该文件。注意 Click 和 CheckedChanged 事件是相同的，因为它们都在用户单击菜单项时处理发生的事件，但菜单项的操作并不相同，应根据菜单项的作用来确定。

16.2 工具栏

通过菜单可以访问应用程序中的大多数功能，把一些菜单项放在工具栏中和放在菜单中有相同的作用。工具栏提供了单击访问程序中常用功能(如 Open 和 Save)的方式。

图 16-1 显示了写字板中的一部分工具栏。

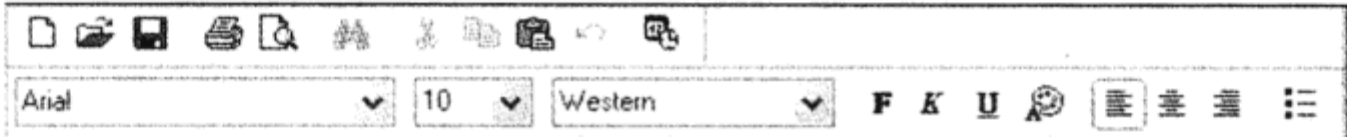


图 16-1

工具栏上的按钮通常包含图片，不包含文本，但它可以既包含图片又包含文本。例如 Word 中的工具栏按钮就不包含文本(参见图 16-1)。包含文本的工具栏按钮有 Internet Explorer 中的工具栏。除了按钮之外，工具栏上偶尔也会有组合框和文本框。如果把鼠标指针停留在工具栏的一个按钮上，就会显示一个工具提示，给出该按钮的用途信息，特别是只显示图标时，这是很有帮助的。

ToolStrip 与 MenuStrip 一样，也具有专业化的外观和操作方式。在用户查看工具栏时，希望能把它移动到自己希望的任意位置上。ToolStrip 就允许用户这么做。

第一次把 ToolStrip 添加到窗体的设计界面上时，它看起来非常类似于前面的 MenuStrip，但存在两个区别：ToolStrip 的最左边有 4 个垂直排列的点，这与 Visual Studio 中的菜单相同。这些点表示工具栏可以移动，也可以停靠在父应用程序窗口中。第二个区别是在默认情况下，工具栏显示的是图像，而不是文本，所以工具栏上项的默认控件是按钮。工具栏显示的下拉菜单允许选择菜单项的类型。

ToolStrip 与 MenuStrip 完全相同的一个方面是，Action 窗口包含 Insert Standard Items 链接。单击这个链接，不会得到与 MenuStrip 相同的菜单项数，而会获得 New、Open、Save、Print、Cut、Copy、Paste 和 Help 等按钮。下面不像前面那样完成一个完整的示例，而是先介绍 ToolStrip 的一些属性和用于填充它的控件。

16.2.1 ToolStrip 控件的属性

ToolStrip 控件的属性控制和管理着控件的显示位置和显示方式。这个控件是前面介绍的 MenuStrip 控件的基础，所以它们具有许多相同的属性。表 16-4 只列出了几个比较重要的属性，如果需要完整的列表，可参阅 .NET Framework SDK 文档说明。

表 16-4

属 性	说 明
GripStyle	控制 4 个垂直排列的点是否显示在工具栏的最左边。隐藏手柄后，用户就不能移动工具栏了
LayoutStyle	控制工具栏上的项如何显示，默认为水平显示
Items	包含工具栏中所有项的集合
ShowItemToolTip	确定是否显示工具栏上某项的工具提示
Stretch	默认情况下，工具栏比包含在其中的项略宽或略高。如果把 Stretch 属性设置为 true，工具栏就会占据其容器的总长

16.2.2 ToolStrip 的项

在 ToolStrip 中可以使用许多控件。前面提到，工具栏应能包含按钮、组合框和文本框。除了与这些对应的控件之外，工具栏还可以包含其他控件，如表 16-5 所示。

表 16-5

控 件	说 明
ToolStripButton	表示一个按钮。用于带文本和不带文本的按钮
ToolStripLabel	表示一个标签。这个控件还可以显示图像，也就是说，这个控件可以用于显示一个静态图像，放在不显示其本身信息的另一个控件上面，例如文本框或组合框
ToolStripSplitButton	显示一个右端带有下拉按钮的按钮，单击该下拉按钮，就会在它的下面显示一个菜单。如果单击控件的按钮部分，该菜单不会打开
ToolStripDropDownButton	类似于 ToolStripSplitButton，唯一的区别是去除了下拉按钮，代之以下拉数组图像。单击控件的任一部分，都会打开其菜单部分
ToolStripComboBox	显示一个组合框
ToolStripProgressBar	在工具栏上嵌入一个进度条
ToolStripTextBox	显示一个文本框
ToolStripSeparator	前面在菜单示例中见过这个控件，它为各个项创建水平或垂直分隔符

在下面的示例中，要扩展菜单示例，添加一个工具栏。该工具栏将包含工具栏的标准控件和另外 3 个按钮：Bold、Italic 和 Underline。还有一个组合框用于选择字体(注意，这里选择字体的按钮

使用下载代码中的图像)。

试一试：扩展工具栏

按照下面的步骤用工具栏扩展前面的示例：

- (1) 继续使用上一个示例，删除 Format 菜单中使用的 ToolStripMenuItem。选择 Show Help Menu 选项，并按下 Delete 键。然后在它的位置上添加 3 个 ToolStripMenuItem，把它们的 CheckOnClick 属性都改为 true：
  - Bold
  - Italic
  - Underline
- (2) 给窗体添加一个 ToolStrip。在 Actions 窗口中，单击 Insert Standard Items，选择并删除 Cut、Copy、Paste 和 Separator 项。插入 ToolStrip 时，RichTextBox 可能不会正确停靠。此时应把 Dock 改为 none，手工重置控件的大小，以填充窗体。这会把 Anchor 属性改为 Top, Bottom, Left, Right。
- (3) 在工具栏的最后，选择 Button 三次和 Separator 一次，从而创建三个新按钮和一个分隔符(单击 ToolStrip 的最后一项，就会打开这些选项)。
- (4) 创建最后两项，先从下拉列表中选择 ComboBox，再添加一个分隔符，作为最后一个选项。
- (5) 选择 Help 项，把它从当前位置拖动到工具栏的最后一个位置上。
- (6) 前 3 个按钮分别是 Bold、Italic 和 Underline 按钮，按照表 16-6 所示给控件命名。

表 16-6

ToolStrip 按钮	名 称
Bold 按钮	boldToolStripButton
Italic 按钮	italicToolStripButton
Underline 按钮	underlineToolStripButton
ComboBox	fontsToolStripComboBox

- (7) 选择 Bold 按钮，单击 Image 属性中的省略号(...)按钮，选中 Project resource file 单选按钮，单击 Import 按钮。如果已下载了本书的源代码，就可以使用 Chapter16\Toolbars\Images 文件夹下的图像 BLD.ico、ITL.ico 和 UNDRLN.ico。注意 VS 提供的默认扩展名不包含 ICO，所以在浏览图标时，必须从下拉列表中选择 Show All Files。
- (8) 对于 Bold 按钮，选择 BLD.ico 图像。
- (9) 选择 Italic 按钮，把它的图像改为 ITL.ico。
- (10) 选择 Underline 按钮，把它的图像改为 UNDRLN.ico。
- (11) 选择 ToolStripComboBox。在 Properties 面板中，按照表 16-7 所示修改属性。

表 16-7

属 性	值
Items	MS Sans Serif Times New Roman
DropDownStyle	DropDownList

- (12) 把 Bold、Italic 和 Underline 按钮的 CheckOnClick 属性设置为 true。
- (13) 要选择组合框中的初始项，应把下面的代码输入到类的构造函数中：

```
public Form1()  
{  
    InitializeComponent();  
  
    fontsToolStripComboBox.SelectedIndex = 0;  
}
```

- (14) 按下 F5 键，运行示例，打开的对话框如图 16-2 所示。

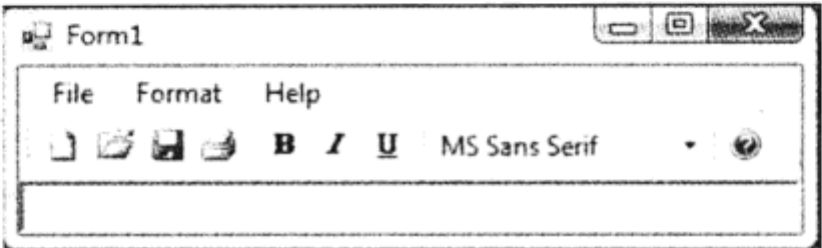


图 16-2

1. 添加事件处理程序

菜单上的 Save、New 和 Open 项已经有了处理程序，显然，工具栏上的按钮应与菜单的操作完全相同。这是很容易实现的，只需给工具栏上按钮的 Click 事件赋予菜单上按钮使用的相同处理程序即可。事件的设置如表 16-8 所示。

表 16-8

ToolStrip 按钮	事 件
New	newToolStripMenuItem_Click
Open	openToolStripMenuItem_Click
Save	saveToolStripMenuItem_Click

下面给 Bold、Italic 和 Underline 按钮添加处理程序。这些按钮都是复选框按钮，所以应使用 CheckStateChanged 事件，而不是 Click 事件。给这 3 个按钮添加该事件。添加如下代码：



可从  
wrox.com  
下载源代码

```
private void boldToolStripButton_CheckedChanged(object sender, EventArgs e)  
{  
    Font oldFont, newFont;  
  
    bool checkState = ((ToolStripButton)sender).Checked;  
    oldFont = this.richTextBoxText.SelectionFont;  
  
    if (!checkState)  
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);  
    else  
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);  
  
    richTextBoxText.SelectionFont = newFont;  
    richTextBoxText.Focus();  
}
```

```
        boldToolStripMenuItem.CheckedChanged -= new
EventHandler(boldToolStripMenuItem_CheckedChanged);
        boldToolStripMenuItem.Checked = checkState;
        boldToolStripMenuItem.CheckedChanged += new
EventHandler(boldToolStripMenuItem_CheckedChanged);
    }
    private void italicToolStripButton_CheckedChanged(object sender,
EventArgs e)
    {
        Font oldFont, newFont;

        bool checkState = ((ToolStripButton)sender).Checked;
        oldFont = this.richTextBoxText.SelectionFont;

        if (!checkState)
            newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
        else
            newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

        richTextBoxText.SelectionFont = newFont;
        richTextBoxText.Focus();

        italicToolStripMenuItem.CheckedChanged -= new
EventHandler(italicToolStripMenuItem_CheckedChanged);
        italicToolStripMenuItem.Checked = checkState;
        italicToolStripMenuItem.CheckedChanged += new
EventHandler(italicToolStripMenuItem_CheckedChanged);
    }

    private void UnderlineToolStripButton_CheckedChanged(object sender,
EventArgs e)
    {
        Font oldFont, newFont;

        bool checkState = ((ToolStripButton)sender).Checked;
        oldFont = this.richTextBoxText.SelectionFont;

        if (!checkState)
            newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
        else
            newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

        richTextBoxText.SelectionFont = newFont;
        richTextBoxText.Focus();

        underlineToolStripMenuItem.CheckedChanged -= new
EventHandler(underlineToolStripMenuItem_CheckedChanged);
        underlineToolStripMenuItem.Checked = checkState;
        underlineToolStripMenuItem.CheckedChanged += new
EventHandler(underlineToolStripMenuItem_CheckedChanged);
    }
}
```

代码段 Chapter16\Toolbars\Form1.cs

事件处理程序简单地把正确的样式设置为 RichTextBox 中使用的字体。在这 3 个方法中，最后 3 行代码分别处理菜单中的对应项。第一行从菜单项中删除事件处理程序，以确保下一行代码运行时不触发事件，第二行代码把 Checked 属性的状态设置为与工具栏按钮相同的值。最后，恢复事件处理程序。

菜单项的事件处理程序应只设置工具栏上按钮的 Checked 属性，让工具栏按钮的事件处理程序完成其他任务。为 CheckedChanged 事件添加处理程序，输入下面的代码：

```
private void boldToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
{
    boldToolStripButton.Checked = boldToolStripMenuItem.Checked;
}

private void italicToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
{
    italicToolStripButton.Checked = italicToolStripMenuItem.Checked;
}

private void underlineToolStripMenuItem_CheckedChanged(object sender,
    EventArgs e)
{
    underlineToolStripButton.Checked = underlineToolStripMenuItem.Checked;
}
```

剩下的是让用户从 ComboBox 中选择一个字体系列。每当用户改变 ComboBox 中的选项，就会触发 SelectedIndexChanged 事件，所以为该事件添加处理程序，输入下面的代码：

```
private void fontsToolStripComboBox_SelectedIndexChanged(object sender,
    EventArgs e)
{
    string text = ((ToolStripComboBox)sender).SelectedItem.ToString();
    Font newFont = null;

    if (richTextBoxText.SelectionFont == null)
        newFont = new Font(text, richTextBoxText.Font.Size);
    else
        newFont = new Font(text, richTextBoxText.SelectionFont.Size,
            richTextBoxText.SelectionFont.Style);
    richTextBoxText.SelectionFont = newFont;
}
```

下面运行代码，就可以在工具栏上设置粗体、斜体和下划线文本了。注意选中或取消选中工具栏上的一个按钮，菜单上的对应选项也会选中或取消选中。

### 16.2.3 StatusStrip 控件

Strip 控件系列中的最后一个控件是 StatusStrip。这个控件在许多应用程序中表示对话框底部的一栏，它通常用于显示应用程序当前状态的简短信息，例如，在 Word 中键入文本时，Word 会在状态栏中显示当前的页面、列和行等。

StatusStrip 派生于 ToolStrip，在把这个控件拖放到窗体上时，读者应很熟悉其视图。在 StatusStrip 中可以使用前面介绍的 4 个控件中的 3 个：ToolStripDropDownButton、ToolStripProgressBar 和



ToolStripSplitButton。还有一个控件是 StatusStrip 专用的，即 StatusStripStatusLabel，它也是一个默认项。

16.2.4 StatusStripStatusLabel 的属性

StatusStripStatusLabel 使用文本和图像向用户显示应用程序当前状态的信息。标签是一个非常简单的控件，没有太多属性，虽然表 16-9 中介绍的两个属性不是专门用于标签的，但它们十分有用。

表 16-9

属 性	值
AutoSize	AutoSize 在默认状态下是打开的，这不是非常直观，因为在改变状态栏上标签的文本时，不希望该标签来回移动，除非标签上的信息是静态的，否则总是应把这个属性改为 false
DoubleClickEnable	在这个属性中，可以指定是否引发 DoubleClick 事件。也就是说，用户可以在应用程序的另一个地方修改信息。例如，让用户双击包含 Bold 的面板，在文本中启用或禁用粗体格式

在下面的示例中，要为前面的示例创建一个简单的状态栏。该状态栏包含 4 个面板，其中 3 个显示图像和文本，最后一个只显示文本。

试一试：StatusStrip 控件

按照下面的步骤扩展前面的小型文本编辑器：

- (1) 在 Toolbox 中双击 StatusStrip，把它添加到对话框中。可能需要重置窗体上 RichTextBox 的大小。
- (2) 在 Properties 面板中，单击 StatusStrip 的 Items 属性中的省略号(...)按钮，打开 Items Collection Editor。
- (3) 单击 Add 按钮 4 次，给 StaturStrip 添加 4 个面板。面板的属性设置如表 16-10 所示。

表 16-10

面 板	属 性	值
1	Name	toolStripStatusLabelText
	Text	Clear this property
	AutoSize	False
	DisplayStyle	Text
	Font	Arial; 8.25pt; style=Bold
	Size	259,17
	TextAlign	Middle Left

(续表)

面 板	属 性	值
2	Name	toolStripStatusLabelBold:
	Text	Bold
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	47, 17
	Image	BLD
	ImageAlign	Middle-Center
3	Name	toolStripStatusLabelItalic
	Text	Italic
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	48, 17
	Image	ITL
	ImageAlign	Middle-Center
4	Name	toolStripStatusLabelUnderline
	Text	Underline
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	76, 17
	Image	UNDRLN
	ImageAlign	Middle-Center

(4) 把下面这行代码添加到 `boldToolStripButton_CheckedChanged` 方法最后的事件处理程序中：

```
toolStripStatusLabelBold.Enabled = checkState;
```

(5) 把下面这行代码添加到 `italicToolStripButton_CheckedChanged` 方法最后的事件处理程序中：

```
toolStripStatusLabelItalic.Enabled = checkState;
```

(6) 把下面这行代码添加到 `underlineToolStripButton_CheckedChanged` 方法最后的事件处理程序中：

```
toolStripStatusLabelUnderline.Enabled = checkState;
```

(7) 选择 RichTextBox, 把 TextChanged 事件添加到代码中, 输入如下所示的代码:

```
private void richTextBoxText_TextChanged(object sender, EventArgs e)
{
    toolStripStatusLabelText.Text = "Number of characters: " +
    richTextBoxText.Text.Length;
}
```

运行应用程序, 所创建的对话框如图 16-3 所示。

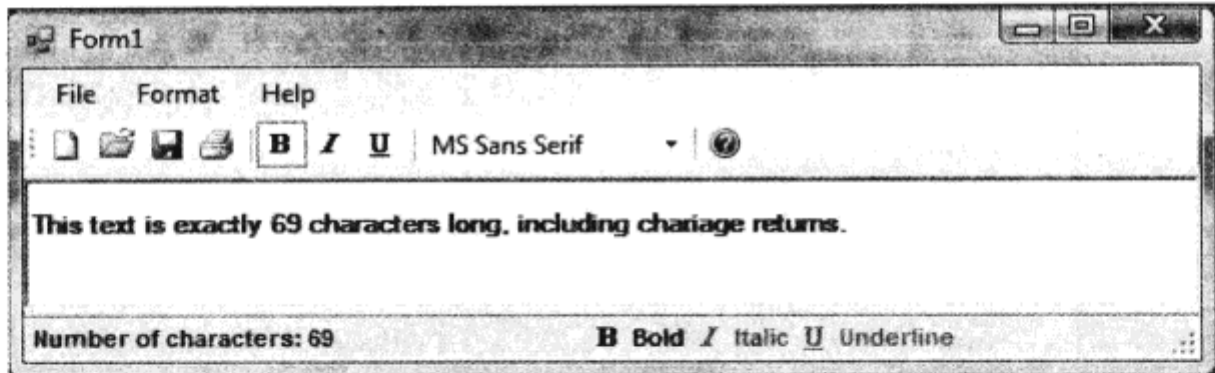


图 16-3

### 16.3 SDI 和 MDI 应用程序

传统上, 可以为 Windows 编写 3 种应用程序, 它们是:

- 基于对话框的应用程序: 它们向用户显示一个对话框, 该对话框提供了所有的功能。
- 单一文档界面 (SDI): 这些应用程序向用户显示一个菜单、一个或多个工具栏和一个窗口, 在该窗口中, 用户可以执行任务。
- 多文档界面 (MDI): 这些应用程序的执行方式与 SDI 相同, 但可以同时打开多个窗口。

基于对话框的应用程序通常用途比较单一, 它们可以完成用户输入量非常少的特定任务, 或者专门处理某一类型的数据。这种应用程序的一个示例是 Windows 中的计算器。

单一文档界面通常用于完成一个特定任务, 因为它允许用户把要处理的单一文档加载到应用程序中。但这个任务通常涉及到许多用户交互操作, 用户也常常希望能保存或加载工作的结果。SDI 应用程序的示例是写字板和画图, 它们都是 Windows 附带的程序。本章前面建立的简单文本编辑器也是 SDI 应用程序。

但一次只能打开一个文档, 所以如果用户要打开第二个文档, 就必须打开一个新的 SDI 应用程序实例, 它与第一个实例没有关系, 对一个实例的任何配置都不会影响第二个实例。例如, 在画图的一个实例中, 可以把绘图颜色设置为红色, 如果打开画图的第二个实例, 绘图颜色仍是默认的黑色。

多文档界面与 SDI 应用程序极为相似, 但它可以在任一时刻在不同的窗口中保存多个已打开的文档。MDI 的标识符包含在菜单栏右边的 Window 菜单中, 该菜单在 Help 的前面。VS 就是一个 MDI 应用程序。VS 的每个设计器和编辑器都在同一个应用程序中打开, 菜单和工具栏会自动调整, 以匹配当前的选择。

16.4 生成 MDI 应用程序

创建 MDI 会涉及到什么问题？首先，希望用户能完成的任务应是需要一次打开多个文档的任务。例如，文本编辑器或文本查看器。第二，应在应用程序中提供工具栏来完成最常见的任务，例如，设置字体样式、加载和保存文档等。第三，应提供一个包含 Window 菜单项的菜单，让用户可以重新定位打开的窗口(平铺和层叠)，显示所有已打开窗口的列表。MDI 应用程序的另一个功能如果是如果打开了一个窗口，该窗口包含一个菜单，则该菜单就应集成到应用程序的主菜单上。

MDI 应用程序至少要由两个截然不同的窗口组成。第一个窗口叫作 MDI 容器(Container)，可以在容器中显示的窗口叫作 MDI 子窗口。MDI 容器既可以叫“MDI 容器”也可以叫“主窗口”，MDI 子容器既可以叫“MDI 子容器”又可以叫“子窗口”。

下面介绍一个小示例，来说明如何完成这些步骤，之后执行更复杂的任务。

试一试：创建一个 MDI 应用程序

创建 MDI 应用程序，首先要像创建其他应用程序那样，在 Visual Studio 中创建一个 Windows 窗体应用程序。

- (1) 在 C:\BegVCSharp\Chapter16 目录中创建一个新的 Windows 应用程序，命名为 MDIBasic。
  - (2) 要把应用程序的主窗口从一个窗体改为 MDI 容器，只需把窗体的 IsMdiContainer 属性设置为 true 即可。改变窗体的背景色，使之表示该窗体现在只有一种背景色，不应放置任何可见的控件(也可以放置控件，在某些情况下这也是合理的，例如创建窗口的停靠区域)。
- 选择窗体，设置如表 16-11 所示的属性。

表 16-11	
属 性	值
Name	frmContainer
IsMdiContainer	True
Text	MDI Basic
WindowState	Maximized

- (3) 要创建子窗口，可以选择 Project | Add New Item，在打开的对话框中选择 Windows Form，给项目添加一个新窗体，命名为 frmChild。
- (4) 把这个新窗体的 MdiParent 属性设置为主窗口的一个引用，该窗体就变成子窗口了。不能通过 Properties 面板设置这个属性，只能通过代码来设置。修改这个新窗体的构造函数：

```
public frmChild(MdiBasic.frmContainer parent)
{
    InitializeComponent();

    MdiParent = parent;
}
```

(5) 在 MDI 应用程序可以按最基本的模式显示之前，还有两件事要做：必须告诉 MDI 容器显示哪个窗口，再显示它们。为此，创建要显示窗体的一个新实例，再对它调用 Show()。要显示为子窗口的窗体的构造函数应与父容器相关联，方法是把它的 MdiParent 属性设置为 MDI 容器的实例。修改 MDI 父窗口的构造函数：

```
public frmContainer()
{
    InitializeComponent();

    frmChild child = new frmChild(this);

    child.Show();
}
```

#### 示例的说明

显示子窗体需要的所有代码放在窗体的构造函数中。首先看看子窗口的构造函数：

```
public frmChild(MdiBasic.frmContainer parent)
{
    InitializeComponent();

    // Set the parent of the form to the container
    this.MdiParent = parent;
}
```

为了把子窗体绑定到 MDI 容器上，子窗体必须注册到容器中。为此，设置该窗体的 MdiParent 属性，如上面的代码所示。注意，使用的构造函数包括参数 parent。

因为 C# 没有为定义了自己构造函数的类提供默认构造函数，所以上面的代码可以防止创建没有绑定到 MDI 容器上的窗体实例。

最后显示窗体，这是在 MDI 容器的构造函数中完成的：

```
public frmContainer()
{
    InitializeComponent();

    frmChild child = new frmChild(this);

    child.Show();
}
```

创建子类的一个新实例，把 this 传递给构造函数，其中 this 表示 MDI 容器类的当前实例。然后对子窗体的新实例调用 Show() 就可以了。如果要显示多个子窗口，只需对每个窗口重复上面代码中突出显示的几行代码即可。

现在运行代码，得到如图 16-4 所示的结果(但 MDI Basic 窗体初始时为最大化窗口，这里重新设置了它的大小，以便放在书页中)。

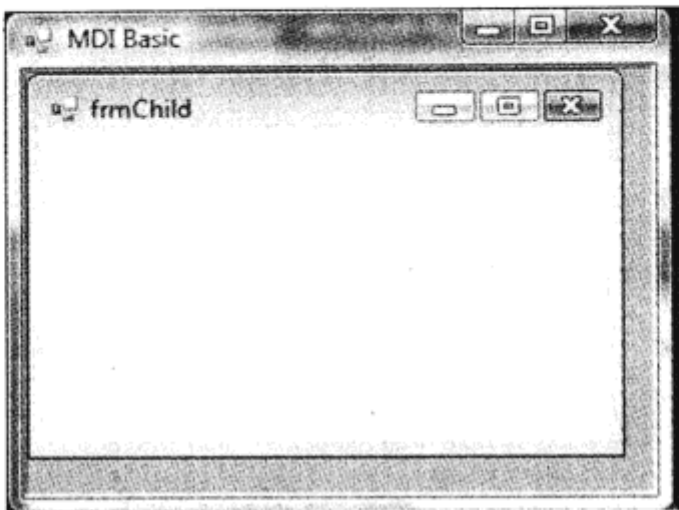


图 16-4

这并不是最吸引人的用户界面，但它是一个好的开端。下一个示例是一个简单的文本编辑器，根据本章前面介绍的菜单、工具栏和状态栏创建。

试一试：创建一个 MDI 文本编辑器

下面先创建一个基本项目，再讨论其中的内容：

- (1) 返回前面的状态栏示例，把窗体重新命名为 `frmEditor`，把它的 `Text` 属性改为 `Editor`。
- (2) 给项目添加一个新窗体 `frmContainer.cs`，在窗体上设置下述属性，如表 16-12 所示。

表 16-12

属 性	值
Name	frmContainer
IsMdiContainer	True
Text	Simple Text Editor
WindowState	Maximized

- (3) 打开 `Program.cs` 文件，在 `Main` 方法中修改包含 `Run` 语句的代码行，如下所示：

```
Application.Run(new frmContainer());
```

- (4) 修改 `frmEditor` 窗体的构造函数：

```
public frmEditor(frmContainer parent)
{
    InitializeComponent();

    this.toolStripComboBoxFonts.SelectedIndex = 0;
    MdiParent = parent;
}
```

- (5) 把菜单项 `&File` 的 `MergeAction` 属性改为 `Replace`，把 `&Format` 菜单项的 `MergeAction` 属性改为 `MatchOnly`。

把工具栏的 `AllowMerge` 属性改为 `False`。

- (6) 给 `frmContainer` 窗体添加一个 `MenuStrip`。再在 `MenuStrip` 中添加一个菜单项 `&File`。

(7) 修改窗体 frmContainer 的构造函数:

```
public frmContainer()  
{  
    InitializeComponent();  
  
    frmEditor newForm = new frmEditor(this);  
    newForm.Show();  
}
```

现在运行应用程序，得到如图 16-5 所示的结果。

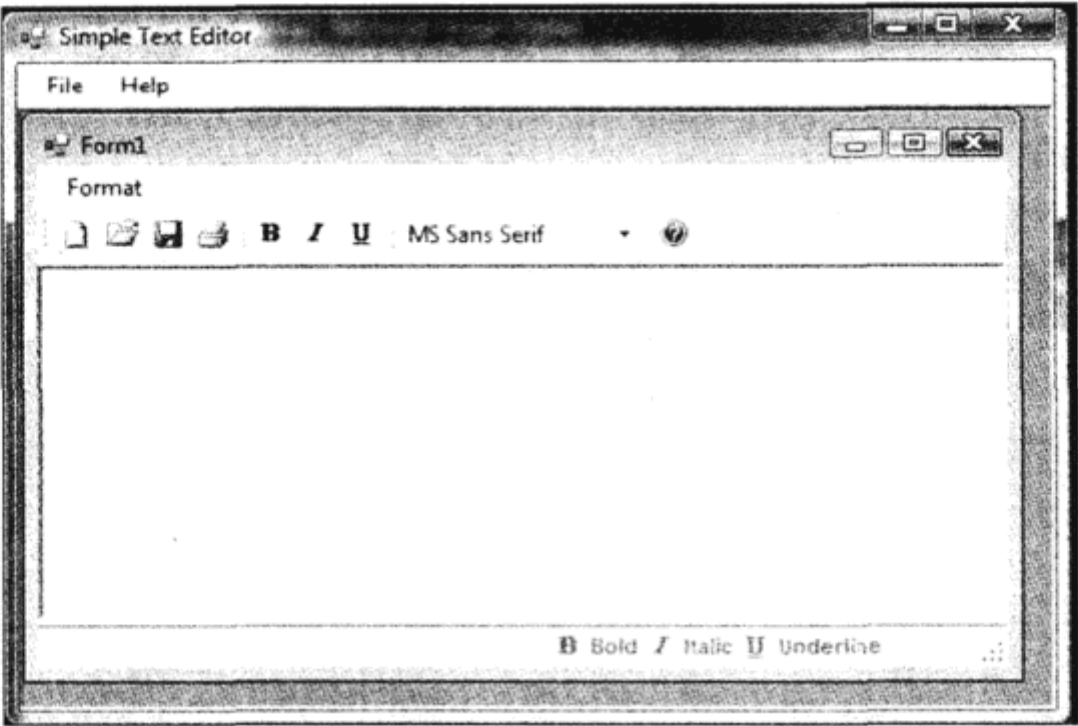


图 16-5

示例的说明

注意这里有一些技巧。File 和 Help 菜单似乎从 frmEditor 窗体中删除了。如果在容器窗口中选择 File 菜单，就会找到 frmEditor 对话框中的菜单项。

如果菜单应包含在子窗口中，则菜单就是这个窗口所特有的。File 菜单在所有窗口中都应该有，不应只包含在子窗口中。原因很明显，如果关闭编辑器窗口，File 菜单就不包含任何菜单项。我们希望能在 File 菜单中插入一些菜单项，当子窗口获得焦点时，这些菜单项是专用于该子窗口的，而其他菜单项在主窗口中显示。

控制菜单项的操作的属性如表 16-13 所示。

表 16-13

属 性	说 明
MergeAction	<p>这个属性指定一个菜单项与另一个菜单合并时该如何操作。可能的值有：</p> <p>Append: 该菜单项放在菜单的最后一个位置上</p> <p>Insert: 插入到满足条件的菜单项的前一位置，该条件可以是菜单项上的文本或菜单项的索引</p> <p>MatchOnly: 需要匹配，但不插入菜单项</p> <p>Remove: 删除满足条件的菜单项，以插入新菜单项</p> <p>Replace: 替换匹配的菜单项，把下拉菜单项添加到新加入的菜单项之后</p>



(续表)

属 性	说 明
MergeIndex	MergeIndex 表示菜单项相对于要合并的其他菜单项的位置。如果要控制所合并菜单项的顺序,就把这个属性设置为大于或等于 0 的值,否则就把它设置为 -1。在进行合并时,会检查这个值,如果它不是 -1,该属性就用于匹配菜单项,而不是文本
AllowMerge	把 AllowMerge 设置为 false 表示不合并菜单

在下面的示例中,要继续完成文本编辑器,修改菜单的合并方式,以反映哪些菜单属于哪个窗口。

试一试：合并菜单

按照下面的步骤修改文本编辑器,以便在容器和子窗口中使用菜单。  
(1) 在 frmContainer 窗体的 File 菜单中添加如下 4 个菜单项,如表 16-14 所示。注意 MergeIndex 值中的跳跃(jump)。

表 16-14

条 目	属 性	值
&New	MergeAction	MatchOnly
	MergeIndex	0
	ShortcutKeys	Ctrl + N
&Open	MergeAction	MatchOnly
	MergeIndex	1
	ShortcutKeys	Ctrl + O
-	MergeAction	MatchOnly
E&xit	MergeAction	MatchOnly
	MergeIndex	11

(2) 需要一种添加新窗口的方式,所以双击菜单项 New,并添加如下代码。这段代码与为显示第一个对话框而输入到构造函数中的代码相同:

```
private void ToolStripMenuItemNew_Click(object sender, EventArgs e)
{
    frmEditor newForm = new frmEditor(this);
    newForm.Show();
}
```

(3) 在 frmEditor 窗体中,从 File 菜单中删除 Open 菜单项,然后修改其他菜单项的属性,如表

16-15 所示。

表 16-15

条 目	属 性	值
&File	MergeAction	MatchOnly
	MergeIndex	- 1
&New	MergeAction	MatchOnly
	MergeIndex	- 1
-	MergeAction	Insert
	MergeIndex	2
&Save	MergeAction	Insert
	MergeIndex	3
Save &As	MergeAction	Insert
	MergeIndex	4
-	MergeAction	Insert
	MergeIndex	5
&Print	MergeAction	Insert
	MergeIndex	6
Print Preview	MergeAction	Insert
	MergeIndex	7
-	MergeAction	Insert
	MergeIndex	8
E&xit	Name	closeToolStripMenuItem
	Text	&Close
	MergeAction	Insert
	MergeIndex	9

(4) 运行应用程序。现在两个 File 菜单已合并了，但子对话框中仍有一个 File 菜单，其中只包含一个菜单项 New。

示例的说明

设置为 MatchOnly 的菜单项不能在菜单之间移动，但对于&File 菜单项，两个菜单项的文本匹配，意味着它们的菜单项合并在一起了。

File 菜单中的菜单项根据其 MergedIndex 属性来合并。其 MergedIndex 属性设置为 MatchOnly 的菜单项保持不变，其他菜单项的 MergedIndex 属性设置为 Insert。

在两个不同菜单上单击菜单项 New 和 Save 时比较有趣。子对话框上的 New 菜单项只是清除文本框的内容，而另一个对话框上的 New 菜单项会创建一个新对话框。这是因为这两个菜单项属于不同的窗口，它们都按照希望的那样工作，但 Save 菜单项如何？单击它，它会从当前的对话框移动到父对话框中。

打开几个对话框，在其中输入一些文本，然后单击 Save 菜单项。打开一个新的对话框，单击

Open 菜单项(Save 菜单项总是保存到同一文件中)。选择其他窗口中的一个，单击 Save，然后返回新对话框，再次单击 Open。结果，Save 菜单项总是跟随有焦点的对话框。每次选择一个对话框时，都会再次合并菜单。

刚才给 frmContainer 对话框的 File 菜单中的 New 菜单项添加了一些代码，创建了对话框。几乎所有 MDI 应用程序都包含 Window 菜单，它允许安排对话框的位置，按照某种方式将其列出。下面的示例就把这个菜单添加到文本编辑器中。

试一试：跟踪窗口

按照下面的步骤扩展应用程序，使之可以显示所有打开的对话框，并排列它们。

- (1) 给 frmContainer 菜单添加一个顶级菜单项&Window。
- (2) 给新菜单添加如表 16-16 中所示的 3 个菜单项。

表 16-16

名 称	文 本
tileToolStripMenuItem	&Tile
cascadeToolStripMenuItem	&Cascade
WindowsSeperatorMenuItem	-

- (3) 选择 MenuStrip 本身，不是选择其中显示的菜单项，将 MDIWindowListItem 属性改为 windowToolStripMenuItem。
- (4) 先双击 Tile 菜单项，再双击 Cascade 菜单项，以添加事件处理程序，然后输入下面的代码：

```
private void ToolStripMenuItemTile_Click(object sender, EventArgs e)
{
    LayoutMdi (MdiLayout.TileHorizontal);
}

private void ToolStripMenuItemCascade_Click(object sender, EventArgs e)
{
    LayoutMdi (MdiLayout.Cascade);
}
```

- (5) 将 frmEditor 对话框的构造函数改为：

```
public frmEditor(frmContainer parent, int counter)
{
    InitializeComponent();

    this.ToolStripComboBoxFonts.SelectedIndex = 0;

    // Bind to the parent.
    this.MdiParent = parent;
    this.Text = "Editor " + counter.ToString();
}
```



(6) 在 `frmContainer` 代码的顶部添加一个私有成员变量, 修改构造函数和菜单项 `New` 的事件处理程序, 如下所示:

```
public partial class frmContainer : Form
{
    private int mCounter;

    public frmContainer()
    {
        InitializeComponent();

        mCounter = 1;
        frmEditor newForm = new frmEditor(this, mCounter);
        newForm.Show();
    }

    private void ToolStripMenuItemNew_Click(object sender, EventArgs e)
    {
        frmEditor newForm = new frmEditor(this, ++mCounter);
        newForm.Show();
    }
}
```

#### 示例的说明

这个示例最有趣的部分是 `Window` 菜单。让一个菜单列出在 MDI 应用程序中打开的所有对话框, 只需在顶级创建一个菜单, 把 `MdiWindowListItem` 属性设置为指向该菜单即可。

然后, `Framework` 就会为当前打开的每个对话框在该菜单的最后追加一个菜单项, 代表当前对话框的菜单项会在旁边显示一个复选标记, 单击该列表中的项可以选择另一个对话框。

其他两个菜单项 `Tile` 和 `Cascade` 演示了窗体的 `MdiLayout` 方法, 该方法允许采用标准方式排列对话框。

对构造函数和 `New` 菜单项的修改只是确保给对话框编号。现在运行应用程序, 添加几个窗口, 注意 `Window` 菜单总是反映当前选中的窗口。

## 16.5 创建控件

有时 `Visual Studio` 提供的控件不能满足用户的需要。原因是多方面的, 控件不能以希望的方式绘制自己, 或者控件在某个方面有限制, 或者需要的控件不存在。为此, `Microsoft` 提供了创建满足需要的控件的方式。 `Visual Studio` 提供了一个项目类型 `Windows Control Library`, 使用它可以创建自己的控件。

可以开发两种不同类型的自定义控件:

- **用户或组合控件:** 这种控件是根据现有控件的功能创建一个新控件。这类控件一般用于把控件的用户界面和功能封装在一起, 或者把几个其他控件组合在一起, 从而改善控件的界面。
- **定制控件:** 当没有控件可以满足要求时, 就创建这类控件, 即从头创建控件。它要自己绘出整个用户界面, 在创建控件的过程中没有现有的控件可以使用。当想要创建的控件的用户界面与其他可用的控件都不同时, 一般需要创建这样的控件。

本章主要讨论用户控件，因为从头设计和绘制定制控件超出了本书的讨论范围。



在 Visual Studio 6 中使用的 ActiveX 控件放在扩展名为 .ocx 的特殊文件中，这些文件实际上是 COM DLL。在 .NET 中，控件存在的方式与其他程序集一样，所以没有 .ocx 扩展名了，控件存在于 DLL 中。

用户控件继承于类 `System.Windows.Forms.UserControl`。这个基类提供的控件具有 .NET 中控件应具有的所有基本功能——用户只需创建控件即可。实际上，任何对象都可以创建一个控件，包括设计俏皮的标签乃至功能全面的网格控件，如图 16-6 所示，底部的框 `UserControl1` 代表一个新控件。

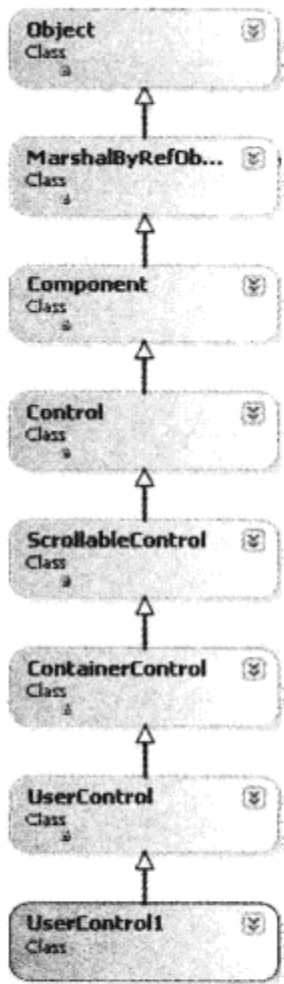


图 16-6



用户控件派生于 `System.Windows.Forms.UserControl` 类，而定制控件派生于 `System.Windows.Forms.Control` 类。

在处理控件时，要考虑几个问题。如果控件不满足这些条件，人们就不会使用它。这些条件是：

- 在设计期间，控件的操作方式应尽可能接近运行期间的操作方式。如果将一个标签和一个文本框合并，创建一个 `LabelTextbox` 控件，标签和文本框就都要在设计期间显示出来，为标签输入的文本也要在设计期间显示出来。在上例中，这是相当简单的，但在比较复杂的情况下，就会出问题，此时需要采取一种折中的方法。

- 应可以在窗体设计器中按合理方式访问控件的属性。例如，ImageList 控件显示了一个对话框，用户可以在该对话框中浏览要包含的图像，导入了图像后，它们就显示在对话框的一个列表中。

在下面的几页中，将通过一个示例说明如何创建控件。这个示例创建 LabelTextbox 控件，并讲述创建用户控件项目、创建属性和事件以及调试控件的基础知识。

从这个控件的名称可以看出，这个控件是使用两个现有控件来创建的。一步执行一个任务在 Windows 编程中非常常见：给窗体添加一个标签，再在该窗体中添加一个文本框，把文本框与标签的位置关联起来。下面看看这个控件的用户可以执行什么操作：

- 用户可以把文本框放在标签的右边或下边，如果文本框放在标签的右边，就可以指定该控件的左边界与文本框之间的固定距离，使文本框对齐。
- 用户应能使用文本框和标签的常用属性和事件。

### 试一试：LabelTextbox 示例

现在知道要做什么了，启动 Visual Studio，并创建一个新项目：

(1) 创建一个新的 Windows Forms Control Library 项目，命名为 LabelTextbox，将其保存在目录 C:\BegVCSharp\Chapter16 中。



如果使用的是 VS 的 Express 版本，就不能使用这个选项。此时应创建一个新的类库项目，在 Project 菜单中把一个用户控件手动添加到项目中。

窗体设计器显示了一个设计界面，它看起来与常用的界面有一些区别。首先，这个界面较小，其次，它看起来根本不像是对话框。读者不应因为这个新界面而泄气，其工作方式是一样的。主要的区别是前面都是把控件放在窗体上，现在则是创建一个要放在窗体上的控件。

(2) 单击设计界面，打开控件的属性。把控件的 Name 属性改为 ctlLabelTextbox。

(3) 双击工具箱中的标签，把它添加到用户控件中，放在设计界面的左上角。把它的 Name 属性改为 lblTextBox，把 Text 属性设置为 Label。

(4) 双击工具箱中的文本框，把它添加到用户控件中，把它的 Name 属性改为 txtLabelText。

在设计期间，不知道用户会如何放置这些控件，所以要编写代码，给标签和文本框定位。这些代码确定了在把 LabelTextbox 控件放在窗体上时控件的位置。

控件的设计是很鼓舞人心的，只是文本框遮挡了一部分标签，其界面也太大了。但这并没有负面作用，因为与习惯使用的其他控件不同，我们看到的并不是最后得到的结果。给该控件添加的代码会改变控件的外观，但只有在把控件添加到窗体时，才改变它的外观。

首先确定控件彼此的相对位置。用户应能决定控件如何定位，为此，要给控件添加两个属性，而不是一个属性。一个属性叫作 Position，允许用户选择两个选项之一：Right 和 Below。如果用户选择了 Right，就使用另一个属性，这个属性叫作 TextboxMargin，是一个 int，表示控件左边界到文本框的像素数。如果用户指定该属性为 0，文本框的右边界就与控件的右边界对齐。

## 1. 添加属性

为了让用户可以选择 **Right** 或 **Below**，先用这两个值定义一个枚举。返回控件项目，进入代码编辑器，添加如下代码：

```
public partial class ctlLabelTextbox : UserControl
{
    public enum PositionEnum
    {
        Right,
        Below
    }
}
```

这是一个第 5 章介绍的一般枚举。这里有一个技巧：要把这个位置设置为一个属性，用户可以通过代码和设计器实现这一设置。为此，给 **ctlLabelTextbox** 类添加一个属性。但首先创建两个成员字段，包含用户选择的值：

```
private PositionEnum mPosition = PositionEnum.Right;
private int mTextboxMargin = 0;
```

然后添加 **Position** 属性，如下所示：

```
public PositionEnum Position
{
    get { return position; }
    set
    {
        position = value;
        MoveControls();
    }
}
```

像添加其他属性那样将这个属性添加到类中。如果要返回该属性，就返回 **position** 成员字段，如果要修改 **Position**，就把值赋给 **position**，并调用 **MoveControls()** 方法。稍后会介绍 **MoveControls()** 方法，现在知道这个方法可以通过检查 **position** 和 **textboxMargin** 的值，来定位两个控件就可以了。

**TextboxMargin** 属性是一样的，但它处理的是一个整数：

```
public int TextboxMargin
{
    get { return textboxMargin; }
    set
    {
        textboxMargin = value;
        MoveControls();
    }
}
```

## 2. 添加事件处理程序

在测试两个属性前，还要添加两个事件处理程序。把该控件放在窗体上时，将调用 **Load** 事件。使用这个事件可以初始化控件和该控件使用的所有资源。处理这个事件是为了移动控件，设置它的大小，使之正好包容它包含的两个控件。



另一个要添加的事件是 `SizeChanged`。每当改变控件大小时，便会引发这个事件。处理这个事件是为了让控件正确地绘制它自己。选择控件，添加两个事件：`SizeChanged` 和 `Load`。

然后添加事件处理程序：

```
private void ctlLabelTextbox_Load(object sender, EventArgs e)
{
    labelCaption.Text = Name;
    Height = textBoxText.Height > labelCaption.Height ?
        textBoxText.Height : labelCaption.Height;
    MoveControls();
}

private void ctlLabelTextbox_SizeChanged(object sender, System.EventArgs e)
{
    MoveControls();
}
```

再调用 `MoveControls()`，定位控件。在再次测试控件之前，先介绍一下这个方法：

```
private void MoveControls()
{
    switch (position)
    {
        case PositionEnum.Below:
            textBoxText.Top = labelCaption.Bottom;
            textBoxText.Left = labelCaption.Left;
            textBoxText.Width = Width;
            Height = textBoxText.Height + labelCaption.Height;
            break;
        case PositionEnum.Right:
            textBoxText.Top = labelCaption.Top;
            if (textBoxMargin == 0)
            {
                int width = Width - labelCaption.Width - 3;
                textBoxText.Left = labelCaption.Right + 3;
                textBoxText.Width = width;
            }
            else
            {
                textBoxText.Left = textBoxMargin + labelCaption.Width;
                textBoxText.Width = Width - textBoxText.Left;
            }
            Height = textBoxText.Height > labelCaption.Height ?
                textBoxText.Height : labelCaption.Height;
            break;
    }
}
```

在 `switch` 语句中测试 `position` 的值，确定应把文本框放在标签的下边还是右边。如果用户选择 `Below`，就把文本框的顶边移动到标签的底边上。然后把文本框的左边界移动到控件的左边界上，把它的宽度设置为控件的宽度。

如果用户选择了 `Right`，就有两种可能性。如果 `TextboxMargin` 是 0，就先确定控件中文本框的宽度，然后把文本框的左边界设置为靠近标签文本的右边界，把剩余的空间设置为文本框的宽度。

如果用户指定了边距，就把文本框的左边界放在该位置上，再次设置宽度。

下面准备测试这个控件，在开始测试前，先生成项目。

### 16.5.1 调试用户控件

调试用户控件与调试 Windows 应用程序大不相同。一般情况下，可以在某个位置添加断点，按下 F5，看看发生什么情况。如果读者仍对调试不熟悉，应参阅第 7 章中的详细论述。

控件需要一个容器来显示它本身，必须提供一个这样的容器，为此，下面的示例创建了一个 Windows Application 项目。

#### 试一试：调试用户控件

(1) 从 File 菜单中选择 Add | New Project，在 Add New Project 对话框中创建一个新的 Windows Application 应用程序，命名为 LabelTextboxTest。这个应用程序只用于测试用户控件，所以最好在 LabelTextbox 项目中创建该项目。

在 Solution Explorer 中，已打开了两个项目。第一个项目是前面创建的 LabelTextbox，以粗体字显示。如果要运行解决方案，调试程序就会把该控件项目用作启动项目。这将会失败，因为控件不是一种独立的项目类型。为了纠正这个问题，右击新项目名 LabelTextboxTest，选择 Set as Startup Project。如果现在运行解决方案，就会运行 Windows 应用程序项目，且不会产生错误。

(2) 现在，在工具箱的顶部应有一个选项卡 LabelTextBox Components。Visual Studio 知道在解决方案中有一个 Windows Control Library，在其他项目也可能使用这个库提供的控件。所以双击控件 ctlLabelTextbox，把它添加到窗体上。注意，Solution Explorer 中的 References 节点被展开了，这是因为 Visual Studio 刚才添加了 LabelTextBox 项目的引用。

(3) 在代码中搜索新的 ctlLabel，应在整个项目中搜索，在后台文件 Form.Designer.cs 中找到它，Visual Studio 把它自动生成的大多数代码都放在这个后台文件中。注意，不能直接编辑这个文件。

(4) 在下面的代码行上放置一个断点。

```
this.MyControl = new LabelTextbox.ctlLabelTextbox();
```

(5) 运行代码，代码会停止在所设置断点的位置上。现在跟踪代码(如果使用默认的键盘映射，就可以按下 F11)。在跟踪代码时，将进入新控件的构造函数，这正是调试组件的地方，也可以设置断点。按下 F5 键，运行应用程序。

### 16.5.2 扩展 LabelTextbox 控件

最后，准备测试控件的属性。注意在把 LabelTextbox 控件添加到窗体上时，其中的控件会移动到正确的位置上。因为把 Position 属性的默认值设置为 Right，所以文本框位于标签的旁边，把 Position 属性改为 Below，文本框会移动到标签之下。

#### 1. 添加更多属性

现在还不能对该控件进行什么操作，因为它还不能改变标签和文本框中的文本。下面添加两个属性：LabelText 和 TextboxText。添加这些属性的方式与添加前两个属性的方式相同，也是打开项目，添加如下代码：

```

public string LabelText
{
    get { return labelCaption.Text; }
    set
    {
        labelCaption.Text = labelText = value;
        MoveControls();
    }
}

public string TextboxText
{
    get { return textBoxText.Text; }
    set
    {
        textBoxText.Text = value;
    }
}

```

还需要声明成员变量 `labelText` 来保存文本:

```

private string mLabelText = "";

public ctlLabelTextbox()
{

```

如果要插入文本,就把文本赋给标签和文本框控件的 `Text` 属性,返回 `Text` 属性的值。如果改变了标签的文本,需要调用 `MoveControls()`,因为标签文本可能会影响文本框的位置。另一方面,插入到文本框中的文本不会使控件移动,如果文本比文本框长,超出文本框的部分就不会显示出来。

最后,必须修改 `Load` 事件:

```

private void ctlLabelTextbox_Load(object sender, EventArgs e)
{
    labelCaption.Text = labelText;
    Height = textBoxText.Height > labelCaption.Height ?
        textBoxText.Height : labelCaption.Height;
    MoveControls();
}

```

`Load` 事件把 `labelCaption` 控件的文本设置为属性的值。这样,设计期间和运行期间显示的文本就是相同的。

## 2. 添加更多事件处理程序

现在该考虑控件应提供的事件了。因为该控件派生于 `UserControl` 类,所以继承了许多无需加以处理的功能。但有许多事件我们不希望以标准的方式交给用户。例如 `KeyDown`、`KeyPress` 和 `KeyUp` 事件。需要修改这些事件的原因是,用户希望在文本框中按下一个键时,就引发这些事件。现在,只有在控件本身获得焦点,且用户按下一个键时,才会引发这些事件。

要改变其操作方式,必须处理文本框引发的事件,把它们发送给用户。给文本框添加 `KeyDown`、`KeyUp` 和 `KeyPress` 事件,并输入下面的代码:

```

private void textBoxText_KeyDown(object sender, KeyEventArgs e)
{
    OnKeyDown(e);
}

```

```

    }

    private void textBoxText_KeyPress(object sender, KeyPressEventArgs e)
    {
        OnKeyPress(e);
    }

    private void textBoxText_KeyUp(object sender, KeyEventArgs e)
    {
        OnKeyUp(e);
    }

```

调用 OnKeyXXX 方法会执行订阅事件的对应方法。

### 3. 添加定制的事件处理程序

在创建一个基类中不存在的事件时，需要做更多的工作。下面创建一个事件 PositionChanged，当 Position 属性改变时，将引发该事件。为了创建这个事件，需要做 3 件事：

- 需要一个合适的委托，用于调用用户赋给事件的方法。
- 用户必须把一个方法赋给事件，以订阅该事件。
- 必须调用用户赋给事件的方法。

要使用的委托是由 .NET Framework 提供的 EventHandler 委托。如第 13 章所述，这是一种特殊的委托，它由其关键字 event 声明。下面的代码声明了一个事件，允许用户订阅该事件：

```

public event System.EventHandler PositionChanged;

public ctlLabelTextbox()
{

```

现在只剩下引发该事件了。当改变 Position 属性时，将引发该事件。所以在 Position 属性的 set 存取器中引发该事件：

```

    public PositionEnum Position
    {
        get { return position; }
        set
        {
            position = value;
            MoveControls();
            if (PositionChanged != null)
                PositionChanged(this, new EventArgs());
        }
    }
}

```

首先，确保检查 PositionChanged 是否为 null，看看有没有订阅者。如果没有，就调用方法。

可以像订阅其他事件那样订阅新的定制事件，但这里有一个小问题：该事件在事件窗口中显示之前，必须先生成控件。只有生成了控件，才能在 LabelTextboxTest 项目的窗体中选择控件，在属性面板的 Events 部分双击 PositionChanged 事件。接着给事件处理程序添加如下代码：

```

private void ctlLabelTextbox1_PositionChanged(object sender, EventArgs e)
{
    MessageBox.Show("Changed");
}

```

该定制事件处理程序什么都不会做，它只是说明位置改变了。  
最后，在窗体上添加一个按钮，双击它，给项目添加该按钮的 Click 事件处理程序，添加如下代码：

```
private void buttonToggle_Click(object sender, EventArgs e)
{
    ctlLabelTextbox1.Position = ctlLabelTextbox1.Position ==
LabelTextbox.ctlLabelTextbox.PositionEnum.Right ?
LabelTextbox.ctlLabelTextbox.PositionEnum.Below :
LabelTextbox.ctlLabelTextbox.PositionEnum.Right;
}
```

当运行应用程序时，就可以在运行期间改变文本框的位置。每次移动文本框，都会触发事件 PositionChanged，显示一个信息框。  
这个示例到此就完成了。还可以细化它，这留给读者作为练习。

16.6 小结

本章继续第 15 章的内容，介绍了 MainMenu 和 Toolbar 控件。本章讨论了如何创建 MDI 和 SDI 应用程序，如何在这些应用程序中使用菜单和工具栏。接着论述如何创建自己的控件，设计该控件的属性、用户界面和事件。第 17 章是讨论 Windows 窗体的收官章节，介绍一种特殊类型的窗体：Windows 常用对话框。

16.7 练习

- (1) 以 LabelTextbox 示例为基础，创建一个新属性 MaxLength，以存储文本框中可以输入的最大字符数，然后创建两个新事件 MaxLengthChanged 和 MaxLengthReached。MaxLengthChanged 事件应在修改 MaxLength 属性时引发，MaxLengthReached 事件应在用户输入一个字符后，使文本框中的文本长度等于 MaxLength 属性值时引发。
  - (2) StatusBar 包含一个属性，允许用户双击状态栏上的一个字段，引发一个事件。修改 StatusBar 示例，允许用户双击状态栏，给文本设置粗体、斜体和下划线样式。确保工具栏、菜单和状态栏上的显示总是同步的：激活粗体样式时，把文本 Bold 改为粗体，斜体和下划线样式亦是如此。
- 附录 A 给出了练习答案。

16.8 本章要点

主 题	重 要 概 念
菜单	使用 MenuStrip 在窗体上显示专业外观的菜单
工具栏	使用 ToolStrip 控件在窗体上显示工具栏
状态栏	StatusStrip 提供了一种显示应用程序当前状态的信息的方式
MDI 应用程序	创建 MDI 应用程序，用于进一步扩展文本编辑器
定制控件	以已有的控件为基础创建自己的控件

# 第 17 章

## 部署 Windows 应用程序

### 本章内容:

- 部署选项概述
- 用 ClickOnce 部署 Windows 应用程序
- 创建 Windows 安装程序的部署软件包
- 用 Windows 安装程序安装应用程序

可以采用多种方式安装 Windows 应用程序,简单的应用程序可以使用简单的 xcopy 部署来安装,但对于上百个客户的安装,xcopy 部署就没那么有用了。在这种情况下,可以使用 ClickOnce 部署,也可以使用 Microsoft Windows 安装程序来安装应用程序。

在 ClickOnce 部署中,可以通过单击某个网站的链接来安装应用程序。如果用户选择在其中安装应用程序的目录,或者如果需要一些注册表项,就应使用 Windows 安装程序部署选项。

本章介绍安装 Windows 应用程序的两个选项。

### 17.1 部署概述

部署就是把应用程序安装到目标系统上的进程。传统上,这种安装是通过调用安装程序来完成的。如果需要在成百上千个客户机上安装,安装过程就非常耗时。为了缓解这个问题,系统管理员可以创建批处理脚本,自动完成安装过程。但是,这仍需要做大量工作来安装和支持不同客户的 PC 和不同版本的操作系统。

由于存在这些问题,虽然 Windows 应用程序可以有更丰富的用户界面,许多公司也把它们的内联网应用程序转换为 Web 应用程序。Web 应用程序只需部署到服务器上,客户机就可以自动获取更新后的用户界面。



编写 Silverlight 应用程序是为多客户应用程序提供基于 Web 的部署的一个选项。



使用 ClickOnce 部署，可以解决在部署 Windows 应用程序时遇到的很多问题。通过单击 Web 页面中的一个链接即可安装应用程序。客户系统上的用户不需要有管理权限，因为应用程序安装在用户特定的目录下。使用 ClickOnce，可以安装带有丰富用户界面的应用程序。应用程序安装在客户机上，所以在安装完成后，不需要保留与客户系统的连接。换言之，应用程序可以脱机使用。这样，应用程序图标位于 Start 菜单上，安全问题更容易解决，应用程序也很容易卸载。

ClickOnce 的一个出色功能是，当客户应用程序启动时，更新过程会自动进行，或者在客户应用程序的运行过程中，更新过程会作为一个后台任务来执行。

但是，ClickOnce 部署也有一些限制：如果需要在全局程序集缓存上安装共享组件，或者应用程序所需的 COM 组件需要注册表设置，或者希望用户来确定安装应用程序的目录，就不能使用 ClickOnce。在这些情况下，必须使用 Windows 安装程序。Windows 安装程序是安装 Windows 应用程序的传统方式。下面先介绍 ClickOnce 部署，再介绍 Windows 安装程序包。

## 17.2 ClickOnce 部署

在使用 ClickOnce 部署时，不需要在客户系统上启动安装程序。客户系统的用户只需单击 Web 页面上的一个链接，应用程序就会自动安装。安装完成后，客户机就可以脱机——客户机不再需要访问从中安装应用程序的服务器。

ClickOnce 安装可以在网站、UNC 共享或文件位置(例如 CD)上进行。利用 ClickOnce，应用程序会安装在客户系统上，可以从 Start 菜单上启动，使用 Add/Remove Programs 对话框卸载。

ClickOnce 部署由清单文件(manifest files)描述。应用程序清单描述了应用程序及其需要的权限。部署清单描述了部署配置信息，如更新策略。在本节的示例中，将为第 16 章创建的 MDI Editor 配置 ClickOnce 部署，而且将再次需要这些代码文件。

### 17.2.1 创建 ClickOnce 部署

在下面的示例中，要修改应用程序的名称，定义有用的程序集设置。

#### 试一试：准备应用程序

- (1) 在 Visual Studio 中打开第 16 章中的 MDI Editor 示例。如果读者没有创建这个示例，可以从 Chapter16Code.zip 中复制完整的文件夹 MDI Editor。使用 Visual Studio 菜单 File | Open | Project/Solution... 打开 MDI Editor 文件夹中的解决方案文件 Manual Menus.sln。
- (2) 在 Solution Explorer 中选择项目的 Properties，再选择 Application 选项卡，如图 17-1 所示。
- (3) 把 Assembly name 改为 MDIEditor。
- (4) 单击 Assembly Information... 按钮。
- (5) 修改 Title、Description、Company、Product 和 Copyright 信息，如图 17-2 所示。
- (6) 选择 Build | Build Solution，生成项目。



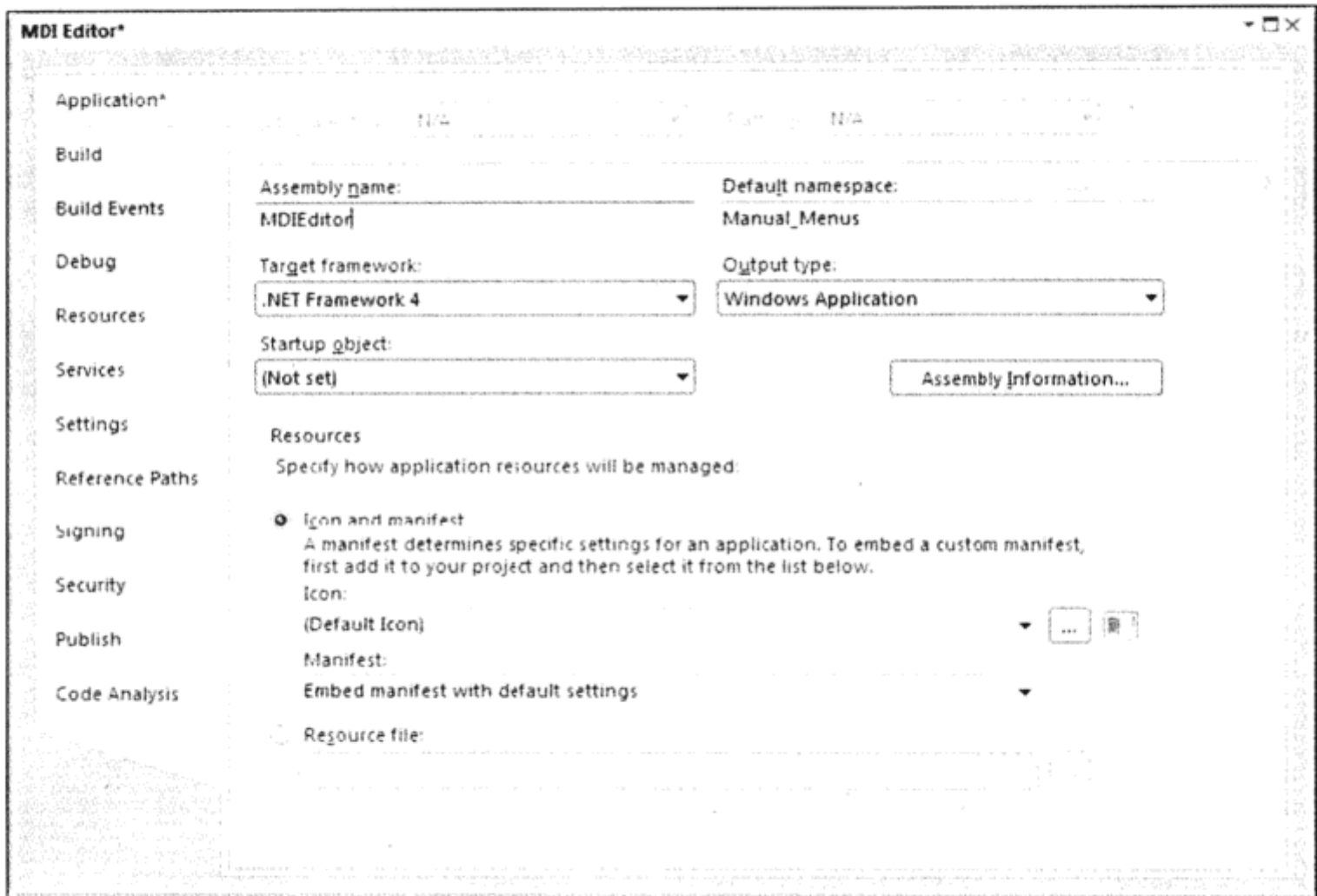


图 17-1

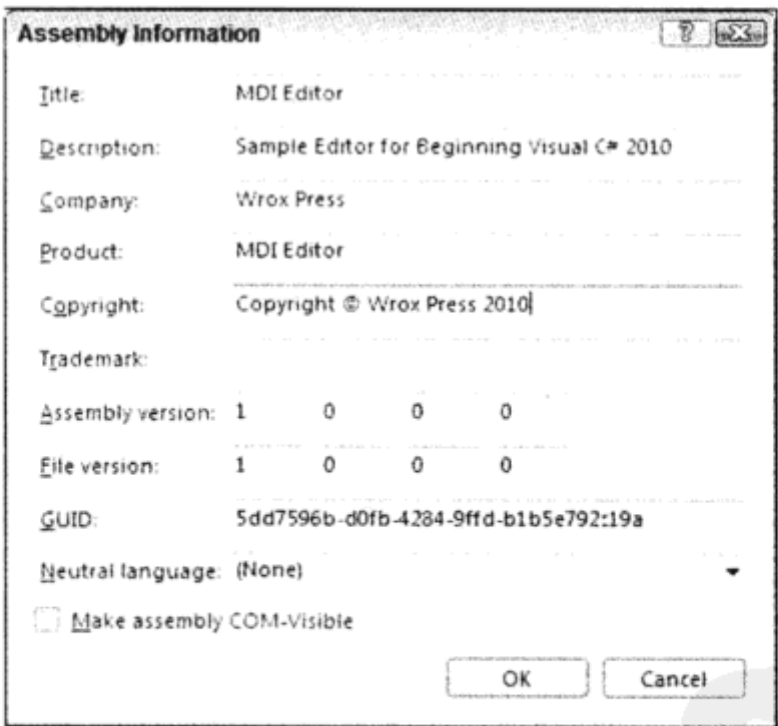


图 17-2

示例的说明

程序集名定义了生成过程中创建的程序集的名称。在安装应用程序时，需要部署这个程序集。通过 Assembly Information 对话框修改的属性改变了 AssemblyInfo.cs 文件中的程序集特性。这些元数据信息由部署工具使用。还可以选择可执行文件，单击菜单中的 Properties，以便从 Windows 资源管理器中读取元数据信息。在 Details 选项卡中可以看到前面添加的信息。

在网络上成功部署程序集，需要使用清单，清单必须有证书签名。该证书会向安装应用程序的用户显示创建安装程序的机构。这样用户就可以确定是否相信该部署。在下面的示例中，要创建一个与 ClickOnce 清单相关的证书。

试一试：签署 ClickOnce 清单

(1) 在 Solution Explorer 中为项目选择 Properties，再选择 Signing 选项卡，如图 17-3 所示。

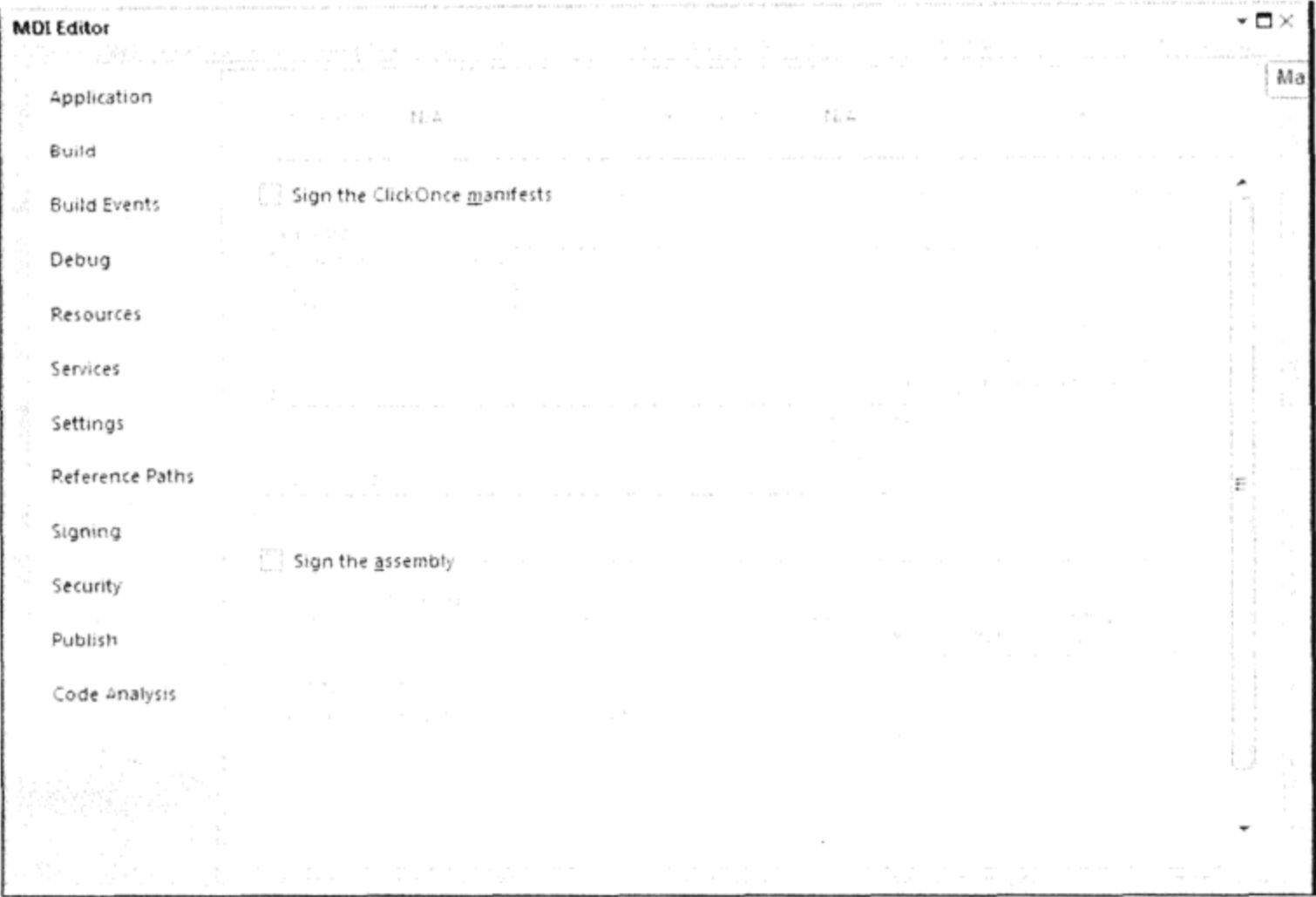


图 17-3

(2) 选中 Sign the ClickOnce manifests 复选框。

(3) 单击 Create Test Certificate...按钮，创建一个与 ClickOnce 清单相关的测试证书。根据要求给证书输入一个密码。必须记住密码，才能在以后进行设置。然后单击 OK 按钮。

(4) 单击 More Details 按钮，可以查看证书信息，如图 17-4 所示。

示例的说明

安装应用程序的用户可以使用证书来辨识安装软件包的创建者。阅读证书的内容，可以确定是否能信任此安装软件包，从而满足安全要求。

创建好测试证书后，用户并没有获得真正值得信任的信息，而是会接收到一个警告，说明这个证书不能信任，如后面所述。这个证书只供测试之用。在应用程序准备好部署之前，必须从证书颁发机构(如 VeriSign)处获得一个真正的证书。如果应用程序只在内联网中部署，还可以从安装在本地网络的本地证书服务器处获得一个证书。Microsoft 证书服务器可以与 Windows Server 2003 或 2008 一起安装。如果有了证书，就可以通过从 Signing 选项卡中单击 Select from File，来配置它。



图 17-4

下面的示例将配置程序集的安全要求。在把程序集安装到客户机上时，必须定义所要求的信任。

试一试：定义安全要求

(1) 在 Solution Explorer 中为项目选择 Properties，再选择 Security，如图 17-5 所示。选中 Enable ClickOnce security settings 复选框。使用默认配置，以完全信任应用程序。

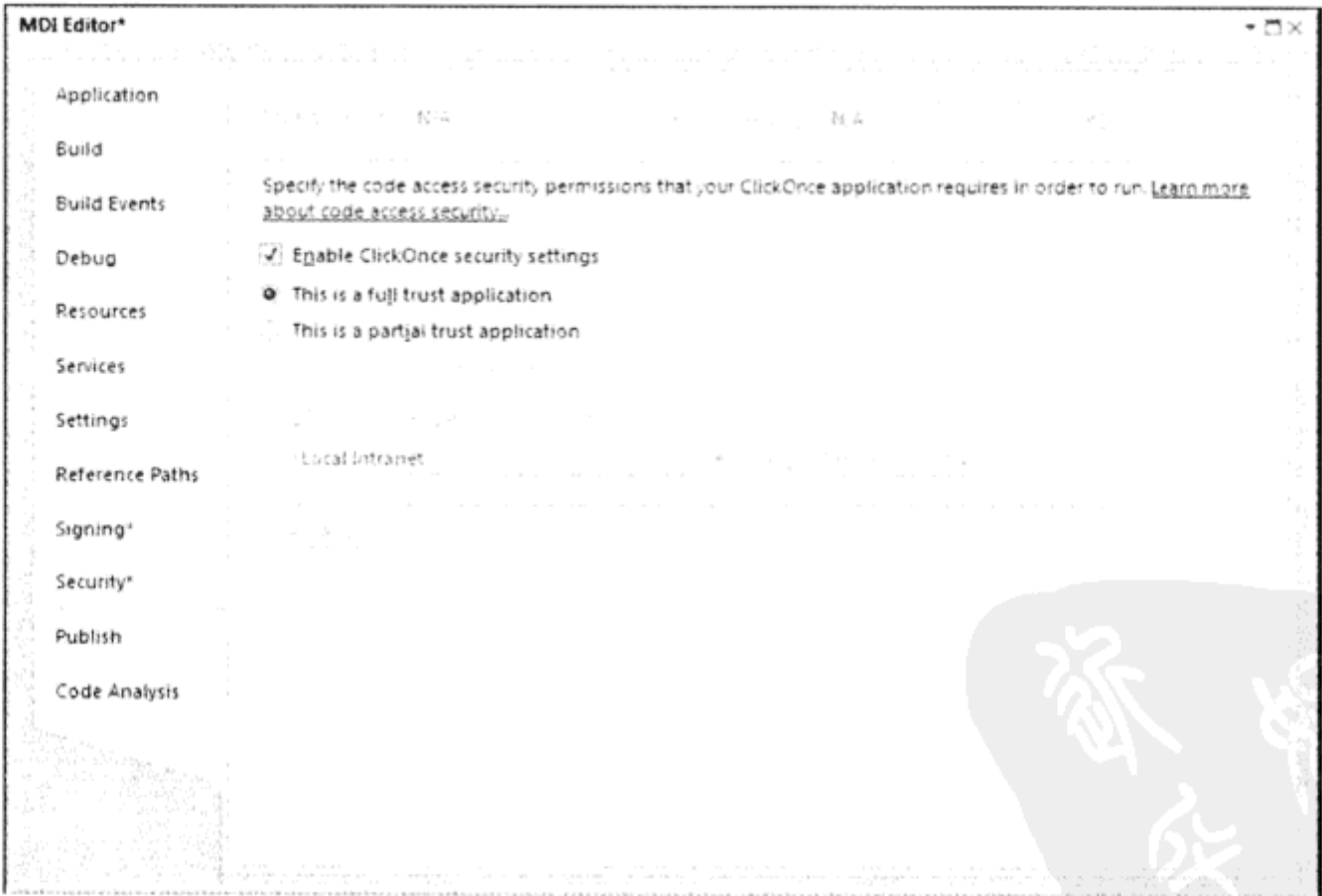


图 17-5

示例的说明

通过 ClickOnce 设置可以把应用程序配置为需要完全信任，或者运行在沙箱中，只能部分信任。

有了完全信任权限，应用程序就可以对系统进行全面的访问，可以执行运行应用程序的用户所允许执行的任何操作。在安装应用程序时，将警告用户：应用程序需要完全信任权限。部分信任的应用程序不能访问文件系统，只能访问独立的存储器或注册表，这种应用程序在沙箱模式下运行。由于 MDI Editor 应用程序需要访问文件系统，所以需要完全信任权限。

定义了安全需求后，就可以开始创建部署清单，以便发布应用程序。使用 Publish 向导很轻松地完成这项任务，如以下示例所示。

试一试：更多的发布配置选项

(1) 选择项目属性中的 Publish 选项卡。单击 Options 按钮，打开 Publish Options 对话框，如图 17-6 所示。选择左边列表中的 Description，输入发布者名称、系列名称、产品名称和支持的 URL。

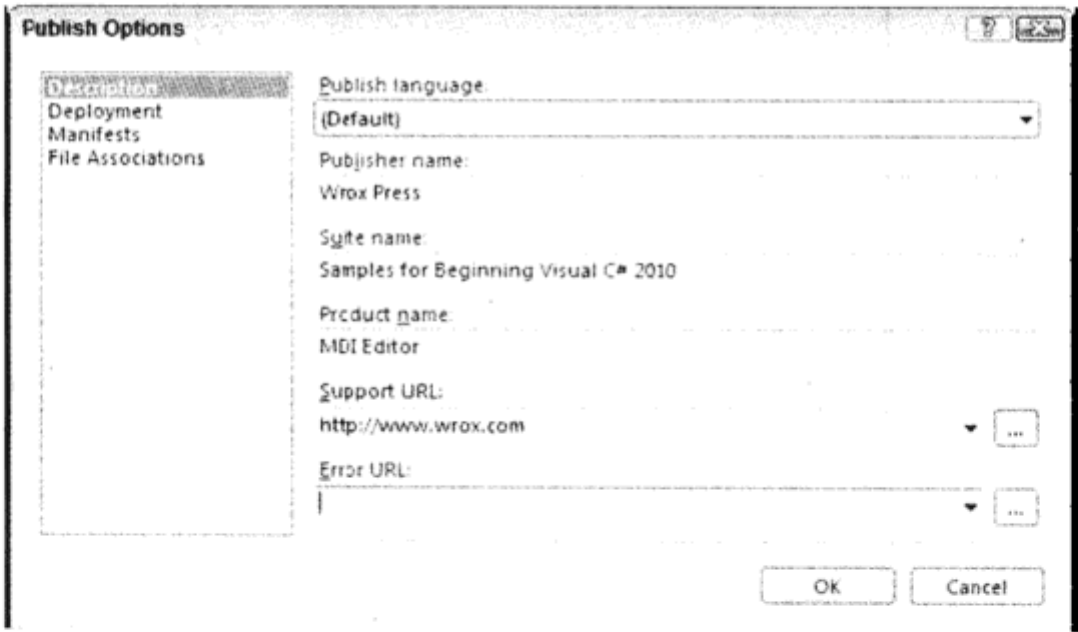


图 17-6

(2) 选择 Updates 按钮来配置 Update 选项，然后选中 The application should check for updates 复选框，如图 17-7 所示。

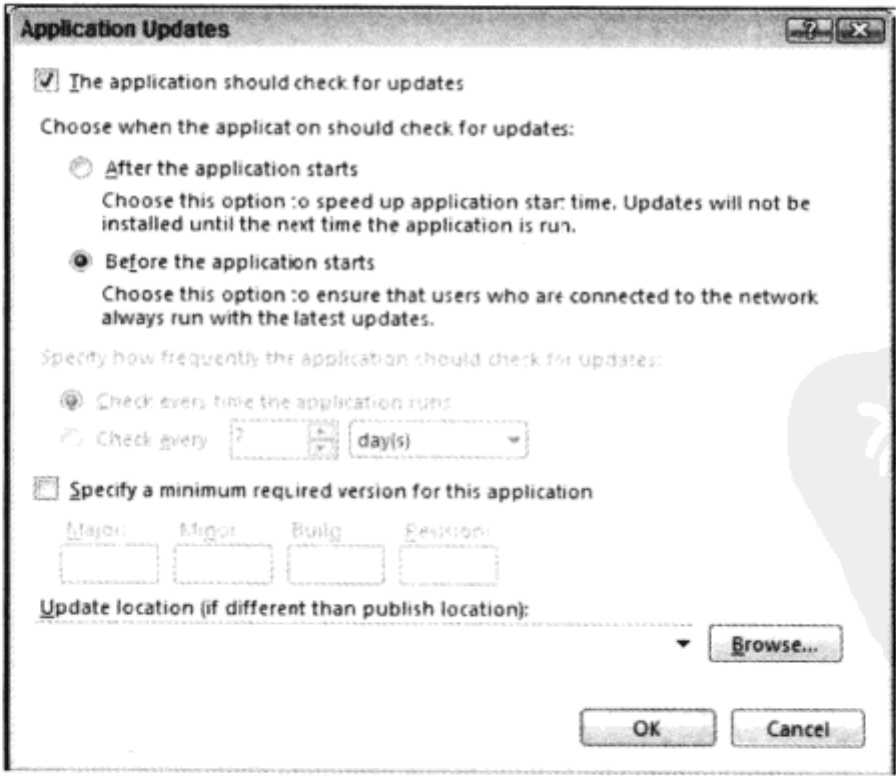


图 17-7

试一试：使用 Publish 向导

(1) 选择 Build | Publish SimpleEditor 菜单，启动 Publish 向导。输入网站的路径 `http://localhost/MDIEditor`，如图 17-8 所示。单击 Next 按钮。

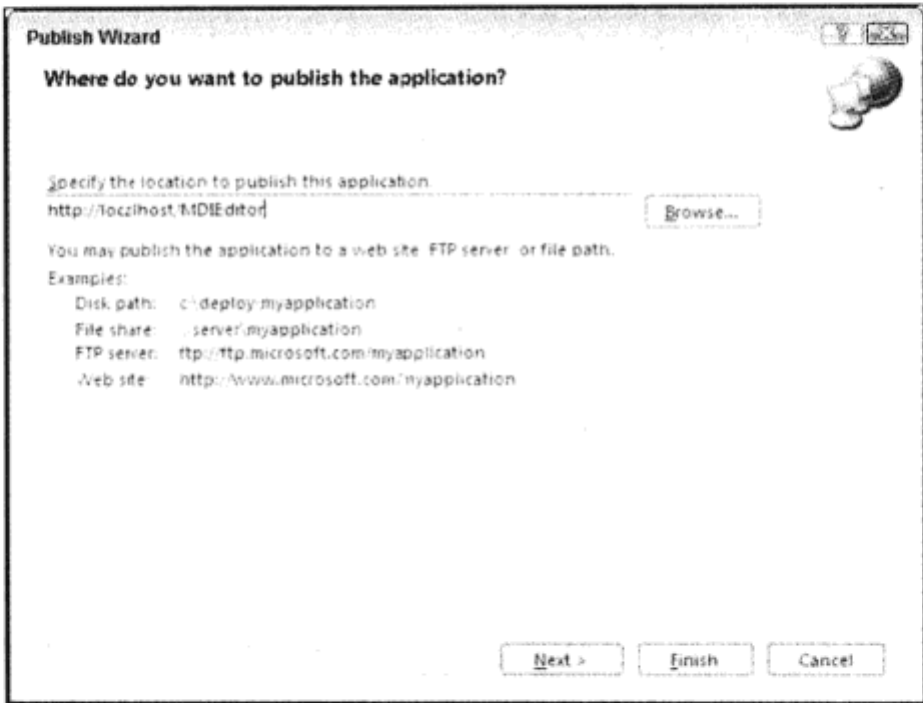


图 17-8

要在 Windows 7 或 Windows Vista 上把应用程序发布到 Web 服务器上，必须以 elevated 模式启动 VS2010，且必须拥有管理权限，还需要安装 IIS。如果没有安装 IIS，应选择发布到本地文件系统上。

(2) 在 Publish 向导的第(2)步，选中 Yes, this application is available online or offline 单选按钮，如图 17-9 所示。单击 Next 按钮。

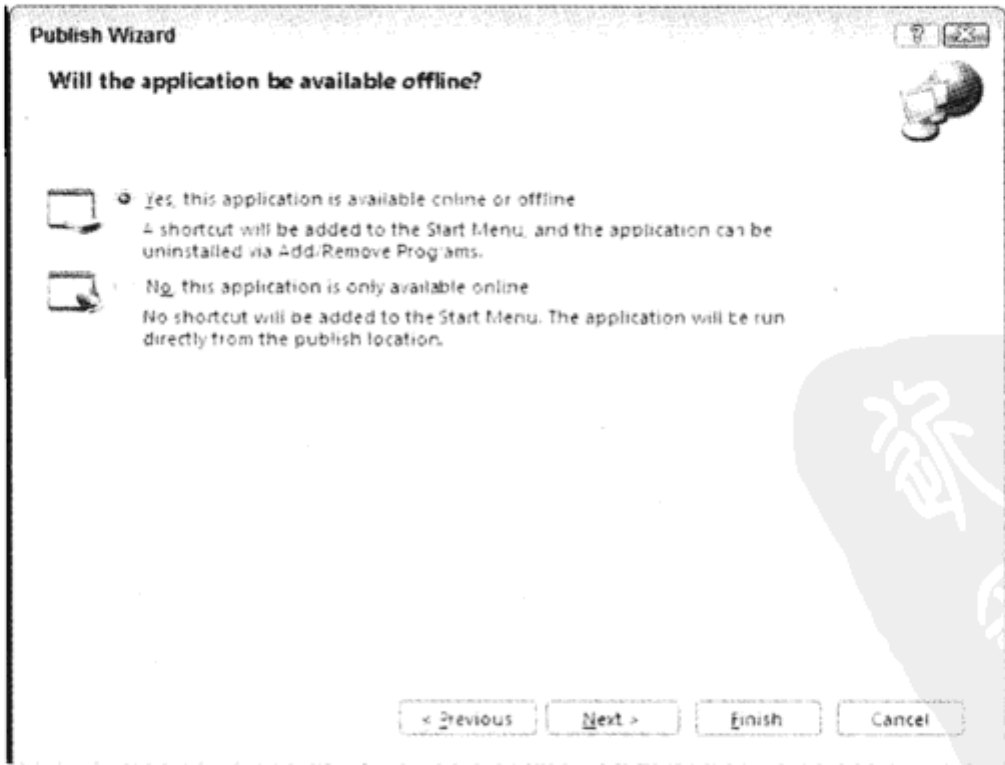


图 17-9

(3) 最后一个对话框给出了准备发布的汇总信息，如图 17-10 所示。单击 Finish 按钮。

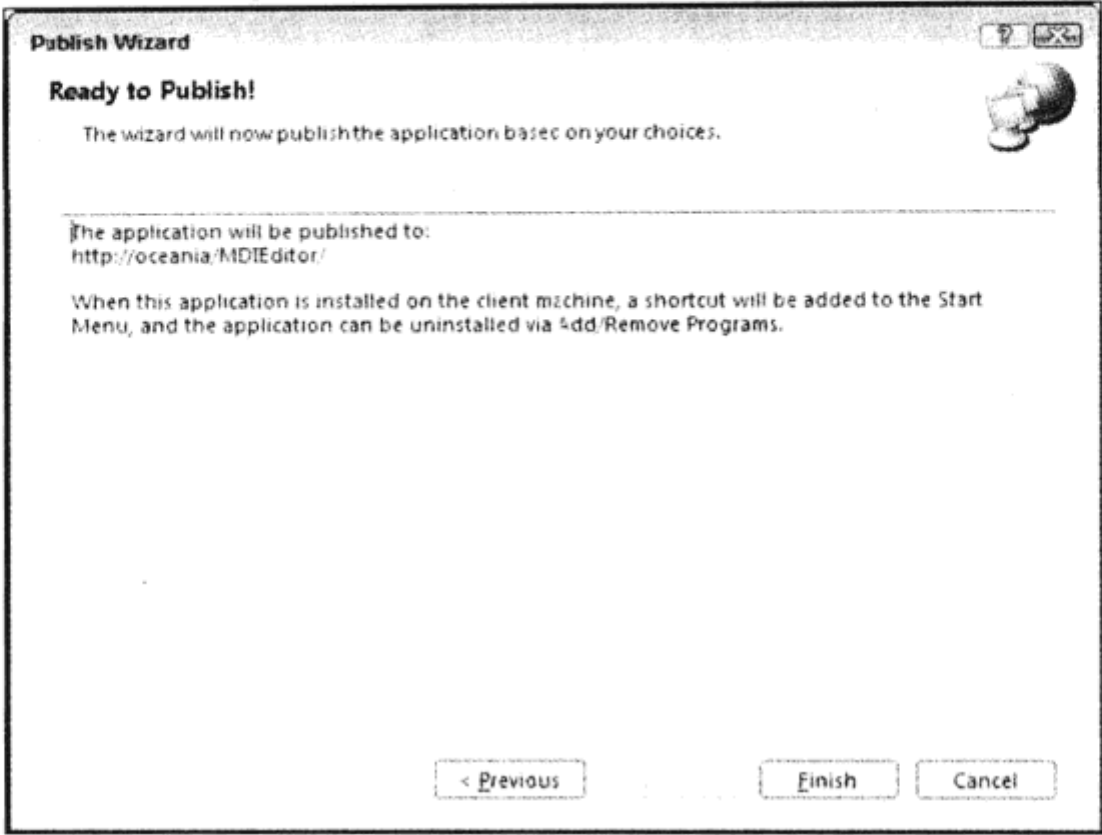


图 17-10

示例的说明

Publish 向导在本地的 Internet Information Services Web 服务器上创建了一个网站。把应用程序的程序集(可执行文件和库)、应用程序和部署清单、setup.exe 和一个示例 Web 页面 publish.htm 复制到 Web 服务器上。部署清单描述了安装信息，如下所示。使用 Visual Studio，可以在 Solution Explorer 中打开 MDIEditor.application 文件，从而打开部署清单。在这个清单中，通过 XML 元素<dependent-Assembly>与应用程序清单建立依赖关系。

```
<deployment install="true"mapFileExtensions="true">
  <subscription>
    <update>
      <beforeApplicationStartup />
    </update>
  </subscription>
  <deploymentProvider
    codebase="http://oceania/MDIEditor/MDIEditor.application" />
</deployment>
<compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">
  <framework targetVersion="4.0" profile="Full" supportedRuntime="4.0.21205" />
</compatibleFrameworks>
<dependency>
  <dependentAssembly dependencyType="install"
    codebase="ApplicationFiles\MDIEditor_1_0_0_0\MDIEditor.exe.manifest"
    size="7416">
    <assemblyIdentity name="MDIEditor.exe" version="1.0.0.0"
      publicKeyToken=" 4e48aff44fcfc18a" language="neutral"
      processorArchitecture="x86" type="win32" />
  </dependentAssembly>
</dependency>
```

```
<dsig:Transforms>
  <dsig:Transform
    Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
</dsig:Transforms>
  <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <dsig:DigestValue>+fiBvjYoMSuDkHZ680iLW2P4y+g=</dsig:DigestValue>
</hash>
</dependentAssembly>
</dependency>
```

选择如图 17-9 所示的选项，以便指定应用程序可以在线和离线使用。这样应用程序会安装在客户系统上，并可以从 Start 菜单中访问。还可以使用 Add/Remove Programs 来卸载应用程序。如果选择应用程序只能在线获得，就必须总是单击网站链接，从服务器上加载应用程序，在本地启动它。

属于应用程序的文件由项目输出定义。单击 Application Files 按钮，就可以在 Publish 设置下看到应用程序文件和应用程序的属性。Application Files 对话框如图 17-11 所示。默认情况下会部署程序集和应用程序清单文件。

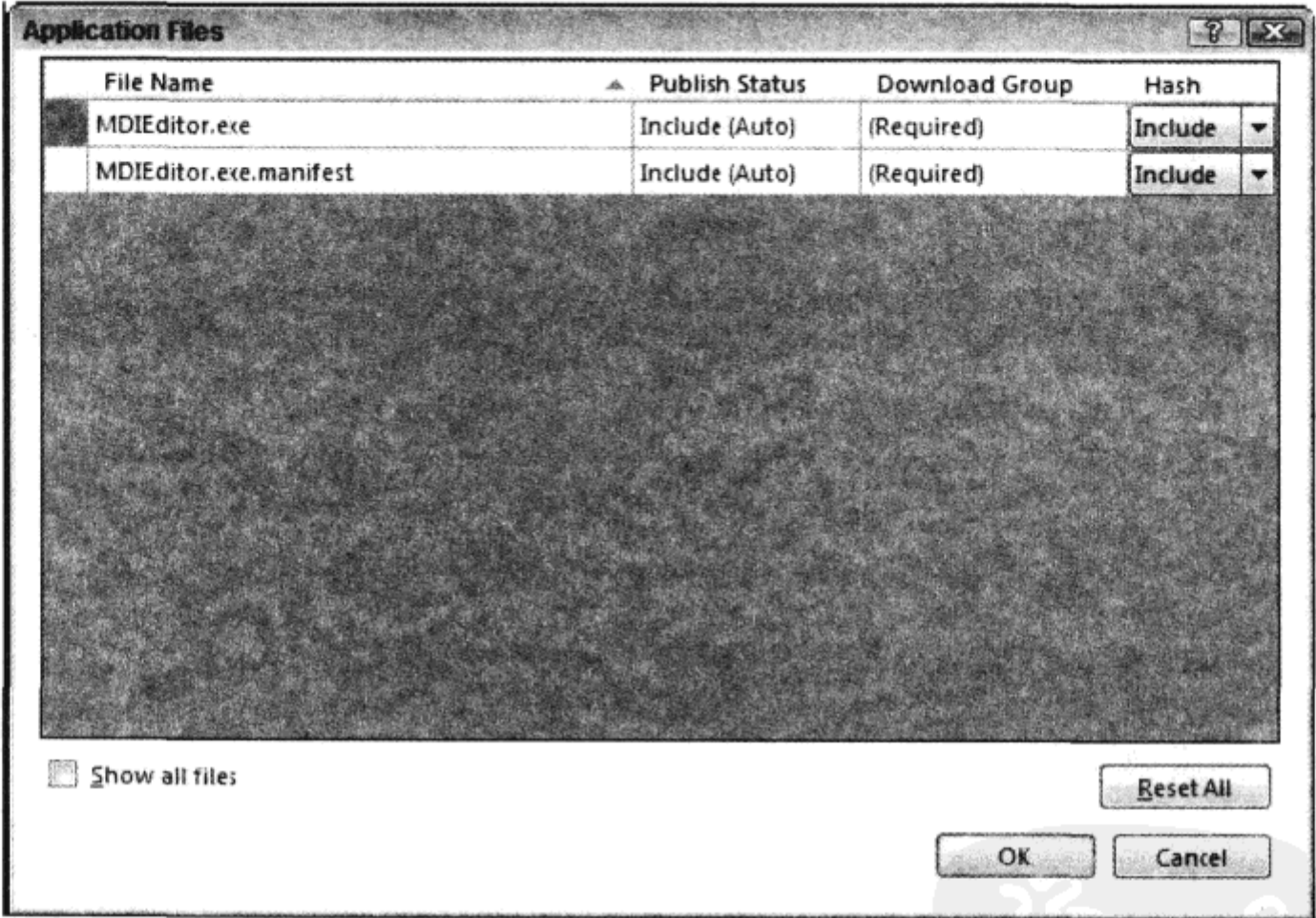


图 17-11

通过 Prerequisites 对话框来定义预先安装的软件包，如图 17-12 所示，单击 Prerequisites 按钮，就可以访问该对话框。对于 .NET 4 应用程序，.NET Framework 4 会自动检测预先安装的软件包，如图 17-12 所示。利用这个对话框还可以选择其他预先安装的软件包。



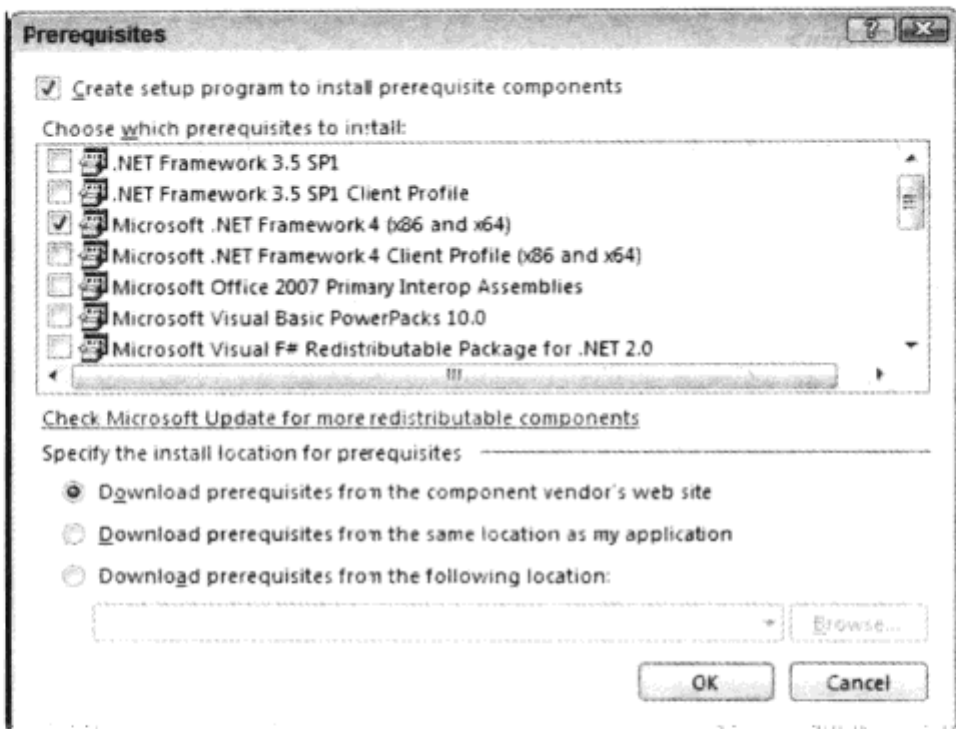


图 17-12



要安装 ClickOnce 应用程序，不需要管理权限。但如果没有在客户系统上安装需预先安装的软件包，就需要管理权限完成安装。

17.2.2 用 ClickOnce 安装应用程序

现在可以按照下面示例中的步骤安装应用程序了。

试一试：安装 MDI Editor 应用程序

(1) 打开 Web 页面 publish.htm，如图 17-13 所示。

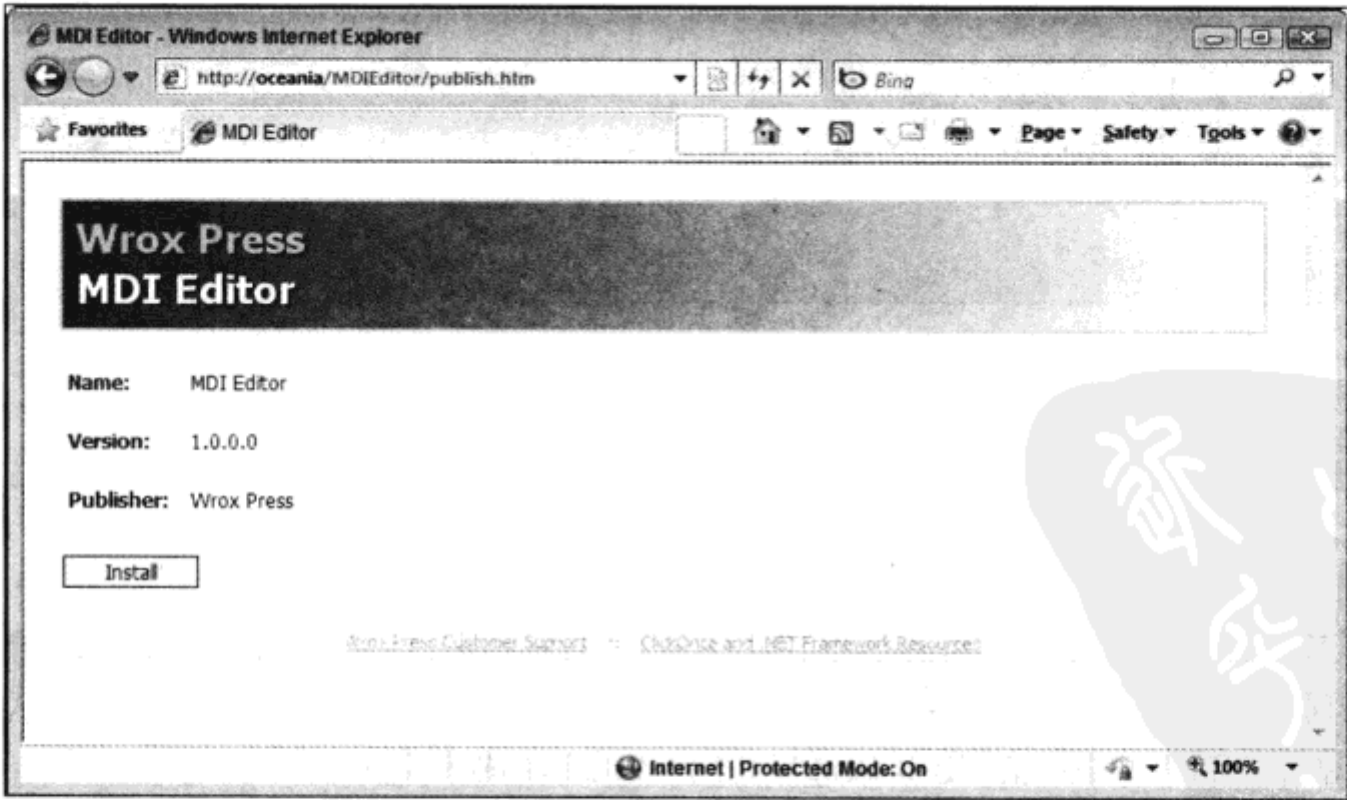


图 17-13

(2) 单击 **Install** 按钮，安装应用程序。此时会弹出一个安全警告，如图 17-14 所示。

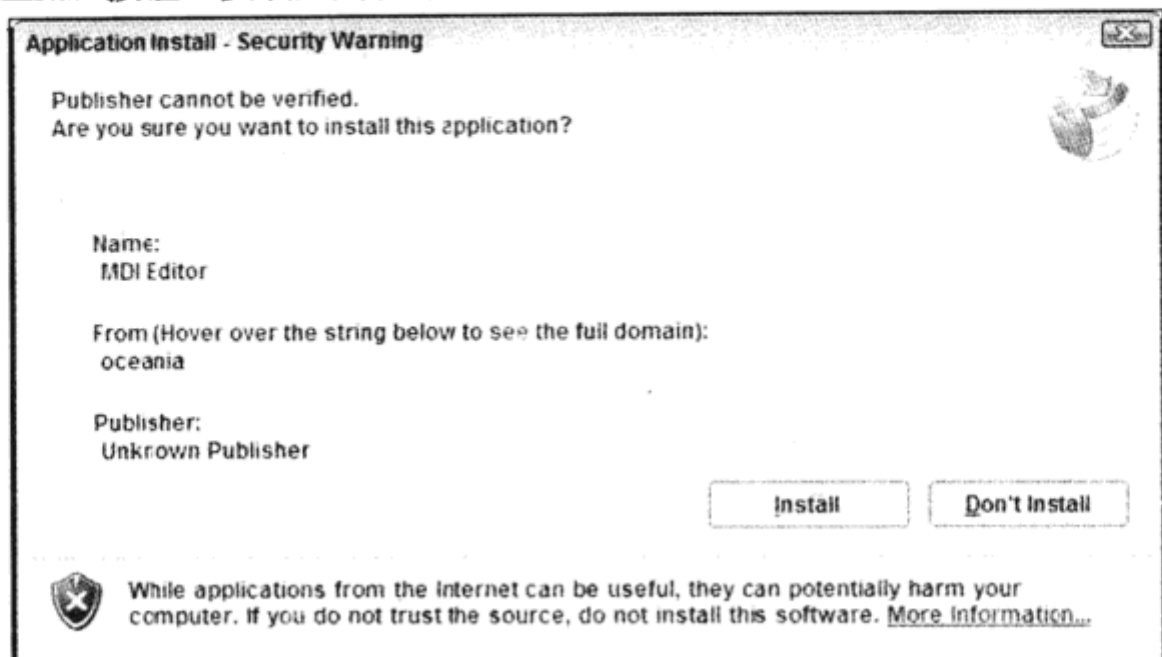


图 17-14

(3) 单击 **More Information...** 链接，查看与应用程序相关的潜在安全问题。阅读此对话框的类别信息，如图 17-15 所示。

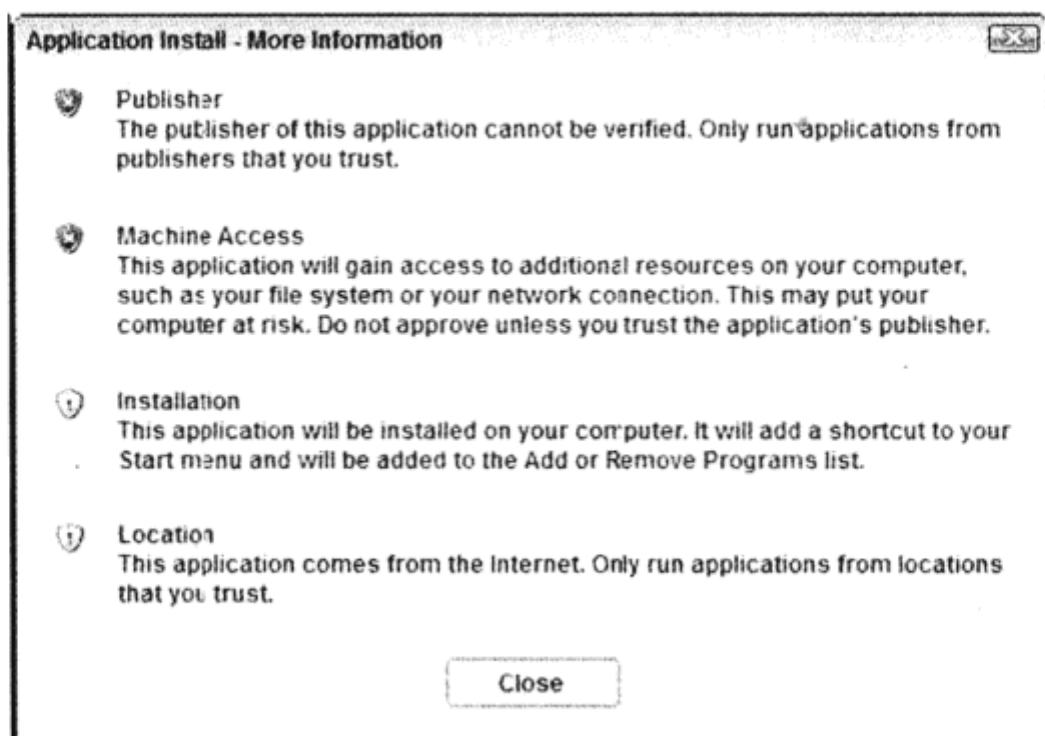


图 17-15

(4) 阅读完对话框的信息后，单击 **Close** 按钮，如果信任所创建的应用程序，就单击 **Application Install** 对话框中的 **Install** 按钮。

#### 示例的说明

打开文件 `publish.htm` 时，会为目标应用程序检查是否存在第 4 版的 .NET 运行库。这个检查由 HTML 页面中的一个 JavaScript 函数来进行。如果运行库未安装，就在安装客户应用程序之前安装运行库。利用默认的发布设置，将从 Microsoft 站点上复制运行库。

单击安装应用程序的链接时，会打开部署清单，以安装应用程序。接着告诉用户应用程序存在的一些可能的安全问题。如果用户单击 **OK** 按钮，就安装应用程序。

17.2.3 创建和使用应用程序的更新包

有了前面配置的更新选项，客户应用程序会自动检查 Web 服务器中是否有新版本。下面的“试一试”示例将对 MDI Editor 应用程序进行这项检查。

试一试：更新应用程序

- (1) 修改 MDI Editor 应用程序，例如，设置 frmEditor.cs 文件中多格式文本框的背景色。
- (2) 在项目属性中选择 Publish 部分，验证发布版本号改为一个新值。
- (3) 生成应用程序，在项目属性的 Publish 部分单击 Publish Now 按钮。
- (4) 不要单击 Web 页面上的 publish.htm 链接，而是从 Start 菜单中启动应用程序。在应用程序启动后，就会弹出如图 17-16 所示的 Update Available 对话框，询问是否要下载新版本。单击 OK 按钮，下载新版本。新版本下载完毕后，就会看到带有彩色多格式文本框的应用程序。

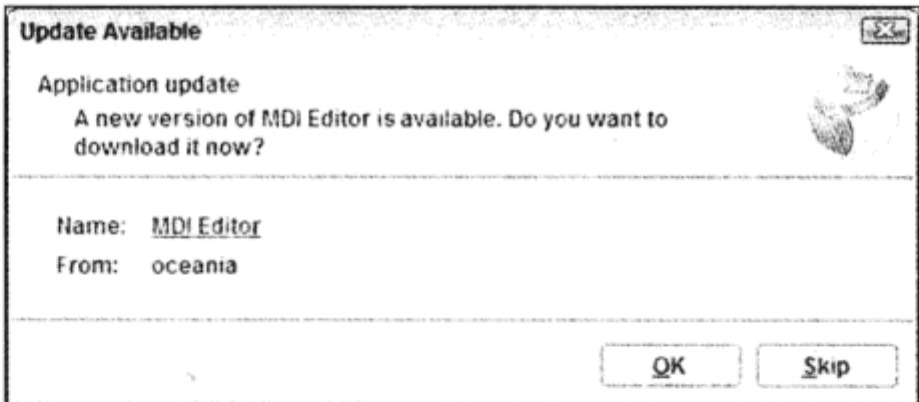


图 17-16

示例的说明

由部署清单中的一个设置和 XML 元素<update>来定义更新策略。使用 Updates 按钮和 Publish 设置可以改变更新策略。一定要使用项目的这些属性来访问 Publish 设置。Application Updates 对话框如图 17-17 所示。

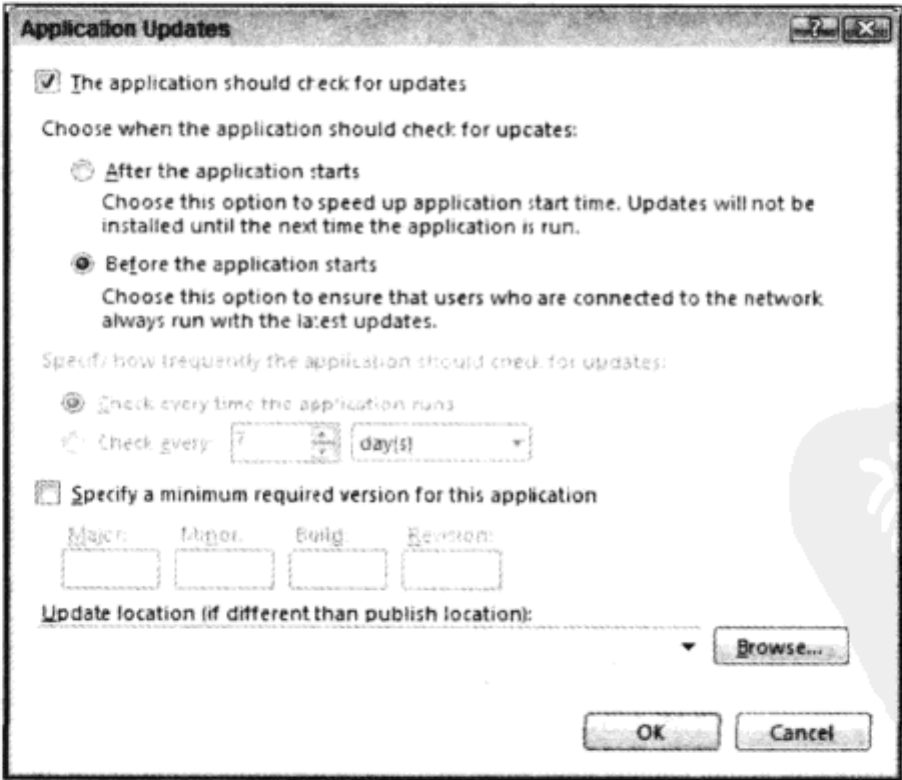


图 17-17

利用这个对话框可以定义客户机是否要查找更新版本。如果要查找更新版本，就可以定义查找是在应用程序启动之前进行，还是在应用程序运行过程中在后台进行更新。如果在后台进行更新，就可以设置更新的时间间隔：是每次启动应用程序时更新，还是每隔特定的小时数、天数或星期数更新一次。

### 17.3 Visual Studio 安装和部署项目类型

使用菜单打开 Visual Studio 的 Add New Project 对话框，从 Other Project Types | Visual Studio Installer 类别的 Installed Templates 窗格中选择 Setup and Deployment 选项之后，就可以看到如图 17-18 所示的窗口。

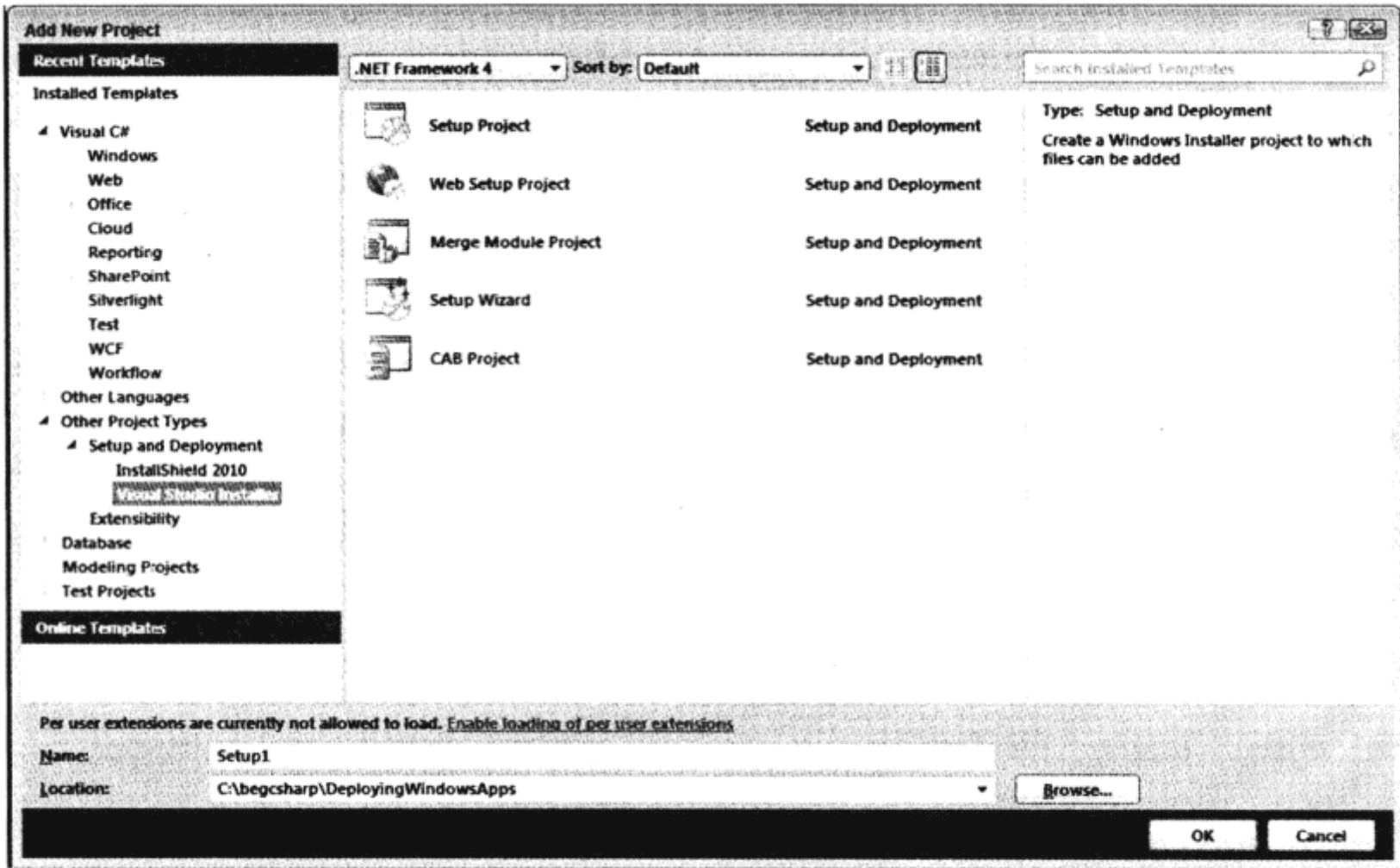


图 17-18

下面是项目类型以及它们的作用：

- 我们要使用 Setup Project 模板。此模板用于创建 Windows 安装软件包，所以可以用来部署 Windows 应用程序。
- Web Setup Project 模板用于安装 Web 应用程序，这个项目模板将在第 20 章使用。
- Merge Module Project 模板用于创建 Windows Installer 合并模块。合并模块(merge module)是安装程序文件，可以包括在多个 Microsoft Installer 安装软件包中。对于随多个安装程序一起安装的组件而言，可以创建一个合并模块，以在安装软件包中包括此模块。合并模块

的一个示例是.NET 运行库本身：它在合并模块中提供，因此可以在应用程序的安装软件包中包括.NET 运行库。在示例应用程序中将要使用一个合并模块。

- Setup Wizard是选择其他模板的一种方式。需要回答的第一个问题是：是不要创建一个安装程序，以安装应用程序或可供重新分发的软件包？根据不同的选择，可以创建 Windows 安装软件包、合并模块或 CAB 文件。
- Cab Project 模板允许创建 cabinet 文件。Cabinet 文件可以用于将多个程序集合并到一个文件中，并进行压缩。因为 Cabinet 文件可以压缩，所以 Web 客户机可以从服务器上下载较小的文件。

## 17.4 Microsoft Windows 安装程序结构

在 Windows Installer 推出之前，程序员必须创建定制的安装程序。生成安装程序要做大量繁琐的工作，而且许多程序并不遵循 Windows 规则。通常要改写旧版本的系统 DLL，因为安装程序不会检查版本。另外，应用程序文件的复制目录通常是错误的。例如，如果使用硬编码的目录字符串 C:\Program Files 但系统管理员改变了默认驱动器号，或使用了操作系统的国际化版本(其中此目录的命名方式各不相同)，安装就会失败。

Windows Installer 的第一个版本随 Microsoft Office 2000 发布，它也可以作为可分发软件包发布，这些可分发软件包可以包含在其他应用程序软件包中。Windows Installer 版本 1.1 新增了对注册 COM+组件的支持，版本 1.2 支持 Windows ME 的文件保护机制。版本 2.0 新增了对.NET 程序集安装和 Windows 64 位版本的支持。而在.NET 4 中，允许使用的 Windows Installer 最低版本是 3.1。

### 17.4.1 Windows 安装程序术语

使用 Windows 安装程序时，必须理解 Windows 安装程序技术使用的一些术语：软件包、功能和组件。



在 Windows 安装程序的环境中，组件与 .NET Framework 使用的术语“组件”不同。Windows 安装程序组件仅仅是一个文件(或者是在逻辑上作为一个整体的多个文件)。这些文件是可执行文件、DLL 或简单的文本文件。

如图 17-19 所示，软件包包含一个或多个功能块。软件包是单一的 Microsoft 安装程序(MSI)数据库。功能是用户眼中的产品功能，由多个特性和组件构成。组件是开发人员从安装的角度来看的；它是最小的安装单元，由一个或多个文件组成。区分功能和组件的原因是单一的组件可以包含在多个功能中，如图 17-19 中的组件 2。一个功能不能包含在多个功能中。

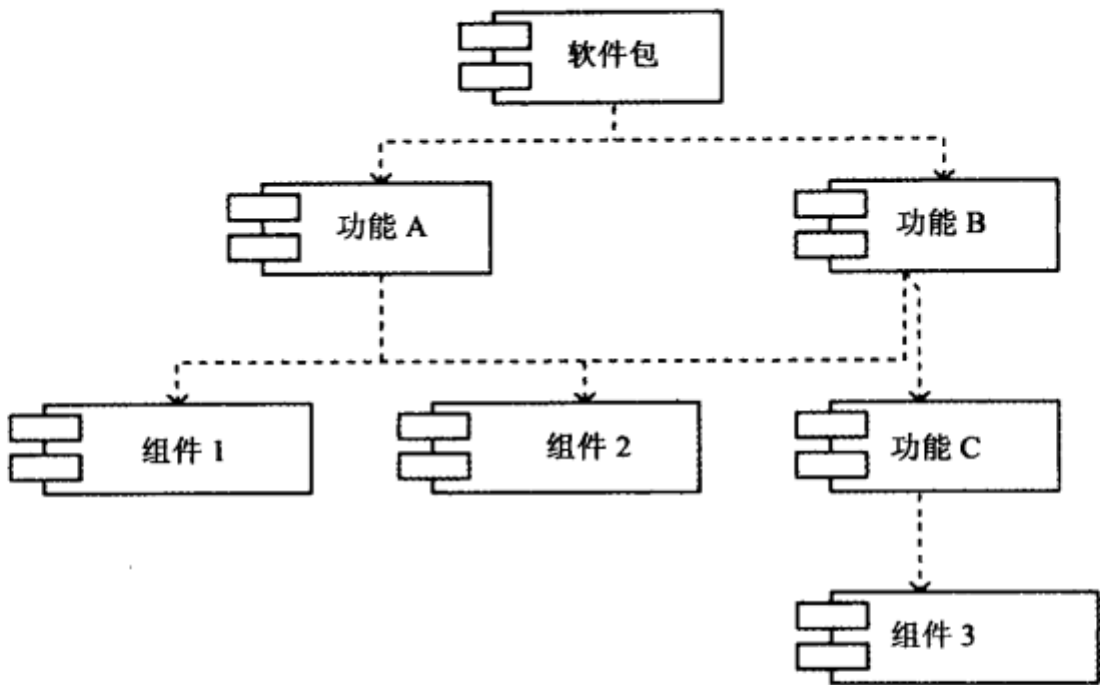


图 17-19

下面介绍一个功能的实际示例：Visual Studio 2010。使用控制面板中的 Programs and Features 选项，单击工具栏上的 Uninstall/Change 按钮，可以在安装之后改变 Visual Studio 的已安装功能，如图 17-20 所示。

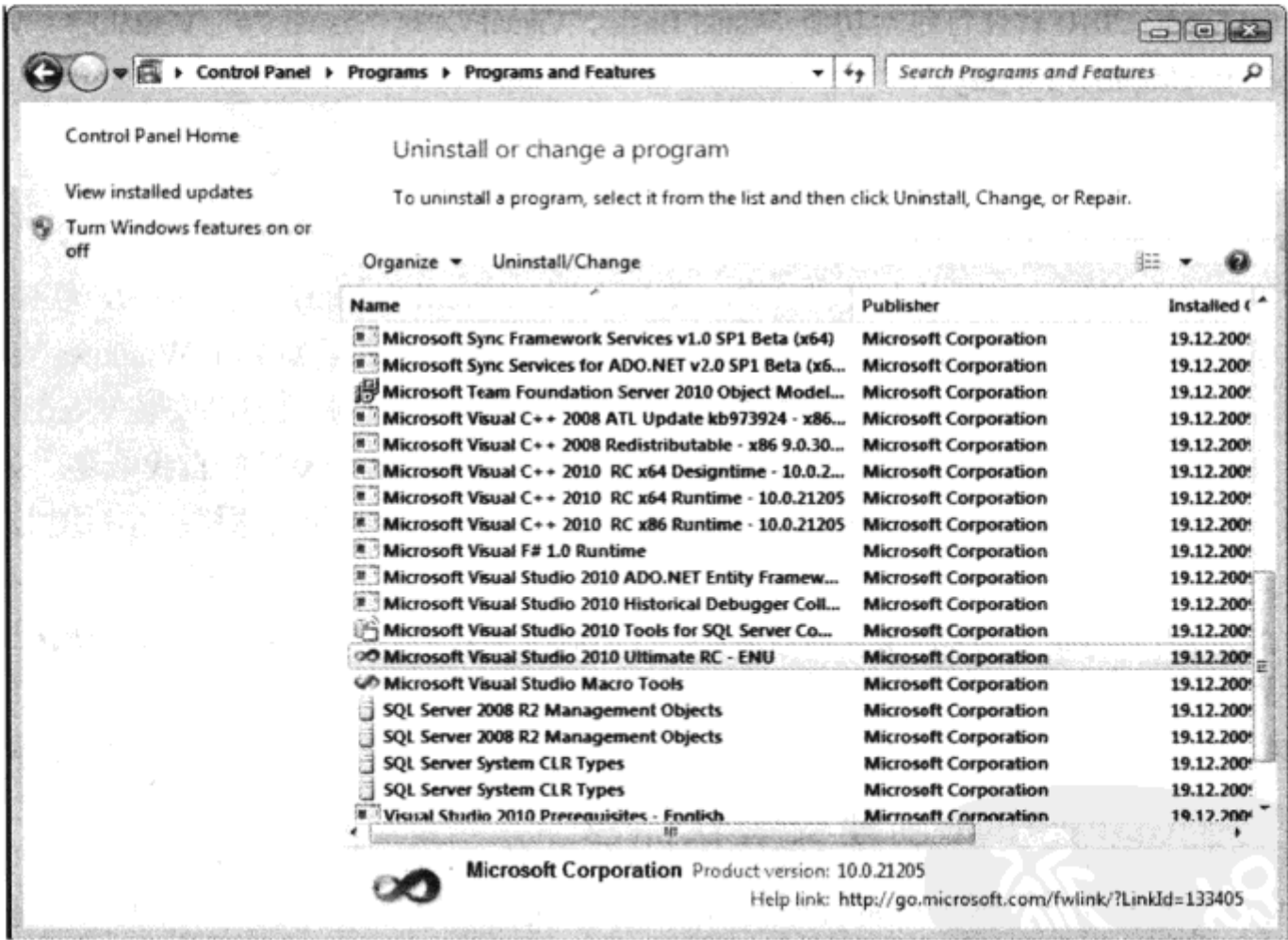


图 17-20

如图 17-21 所示，单击 Uninstall/Change 按钮，可以访问 Visual Studio 2010 Maintenance 向导，这是查看正在使用的功能的有效方法。单击左边树形视图中的加号和减号，就可以查看 Visual Studio 2010 软件包的所有功能。



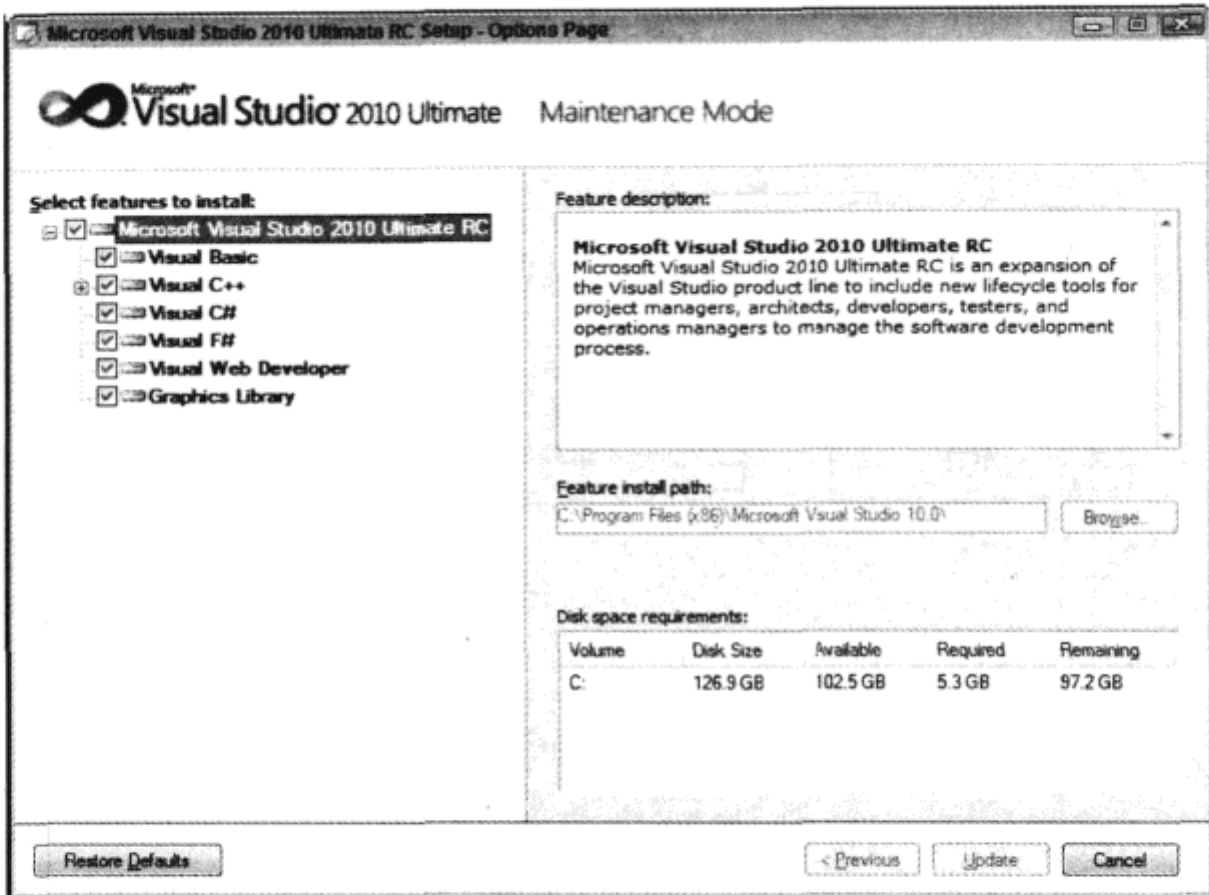


图 17-21

Visual Studio 2010 软件包包括功能 Visual Basic、Visual C++、Visual C#、Visual F#、Visual Web Developer 和 Graphics Library。

17.4.2 Windows 安装程序的优点

Windows 安装程序的优点如下：

- 可以安装功能，也可以不安装功能，或进行通知(advertisement)。有了通知，软件包的功能会在首次使用时安装。或许您在使用 Microsoft Word 时已经注意到了 Windows 安装程序的用法。如果未安装 Word 的通知功能，只要您使用此功能，就会自动安装 Word。
- 如果应用程序受损，可以通过 Windows 安装程序软件包的修复功能自我修复。
- 如果安装失败，就会自动回滚。安装失败之后，所有内容都保持原样：在系统上没有附加的注册表项和文件等。
- 使用卸载功能，可以删除所有的文件，注册表项等。这样就可以完全卸载应用程序。不会留下临时文件，注册表也可以恢复原样。

阅读 MSI 数据库文件的表，可以获取以下信息：复制了什么文件，写入了什么注册表项等。

17.5 为 MDI Editor 创建安装软件包

本节将使用第 16 章的 MDI Editor 解决方案，通过 Visual Studio 2010 创建 Windows 安装软件包。当然，在执行这些步骤时，也可以使用以前开发的其他 Windows 窗体或 WPF 应用程序；只要改变一些名称即可。

17.5.1 规划安装内容

在开始生成安装程序之前，必须规划安装内容。首先要考虑一些问题：



- 应用程序需要什么文件？当然是可执行文件和一些组件程序集。无需标识这些项之间的依赖关系，因为这种依赖关系会自动包括。或许还需要其他一些文件。如文档文件、readme.txt、许可文件、文档模板、图像和配置文件等。我们必须了解所有需要的文件。

对于第 16 章开发的 MDI Editor 应用程序而言，需要可执行文件，还要包含文件 readme.rtf、license.rtf 和显示在安装对话框中的 Wrox Press 位图。

- 应该使用什么目录？应用程序文件应该安装在 Program Files\Application name 中。Program Files 目录的命名因操作系统使用的语言而异。而且，管理员也可以为此应用程序选择不同的路径。无需知道此目录的位置，因为 API 函数调用可以获取此目录。有了此安装程序，我们就可以使用一个预定义的特定文件夹在 Program Files 目录中放置文件。



任何情况下目录都不应该是硬编码的。对于国际版本，这些目录可以有不同的命名！即使应用程序仅支持 Windows 的 English 版本(实际上不会这样)，系统管理员也可以将这些目录移到不同的驱动器中。

MDI Editor 应用程序将可执行文件放在默认的应用程序目录中，除非安装用户选择了另一条路径。

- 用户如何访问应用程序？可以在 Start 菜单中为可执行文件设置快捷方式，在桌面上放置图标等。如果希望在桌面上放置图标，就应该考虑用户是否乐意。对于 Windows XP，其原则是尽可能地使桌面干净。在 Windows 7 中，用户可以在桌面上放置小组件(活动的小程序)，这是桌面应整洁，用户应该根据需要安排图标和 gadget 的一个原因。MDI Editor 应该可以从 Start 菜单上访问。
- 分发介质是什么？希望将安装软件包放在 CD、软盘或网络共享中吗？
- 用户应回答什么问题？用户应接受许可信息，查看 ReadMe 文件，并输入安装路径吗？安装需要其他选项吗？

Visual Studio 2010 Installer 提供的默认对话框足以满足本章后面创建的 Windows Installer 项目的要求。我们会要求用户提供安装程序的目录(用户可以选择不同于默认路径的路径)，显示 ReadMe 文件，并要求用户接受许可协议。

### 17.5.2 创建项目

在了解了安装软件包的内容之后，就可以使用 Visual Studio 2010 安装程序来创建安装程序项目，并添加所有应该安装的文件。下面的示例会使用 Project 向导，配置项目。

#### 试一试：创建 Windows Installer 项目

(1) 打开第 16 章创建的 MDI Editor 项目的解决方案文件。在现有解决方案中添加安装项目。如果没有在第 16 章创建该解决方案，可以从 Chapter16Code.zip 文件中复制完整的文件夹 MDI Editor。在 Visual Studio 中使用 File | Open | Project/Solution 菜单，选择文件夹 MDI Editor 中的解决方案文件 Manual Menus.sln，打开项目。

(2) 添加一个安装项目 MDIEditorSetup：选择 File | Add New Project 菜单，再选择 Other Project Types | Setup and Deployment | Visual Studio Installer，最后选择 Setup Project 模板，如图 17-22 所示，

单击 OK 按钮。

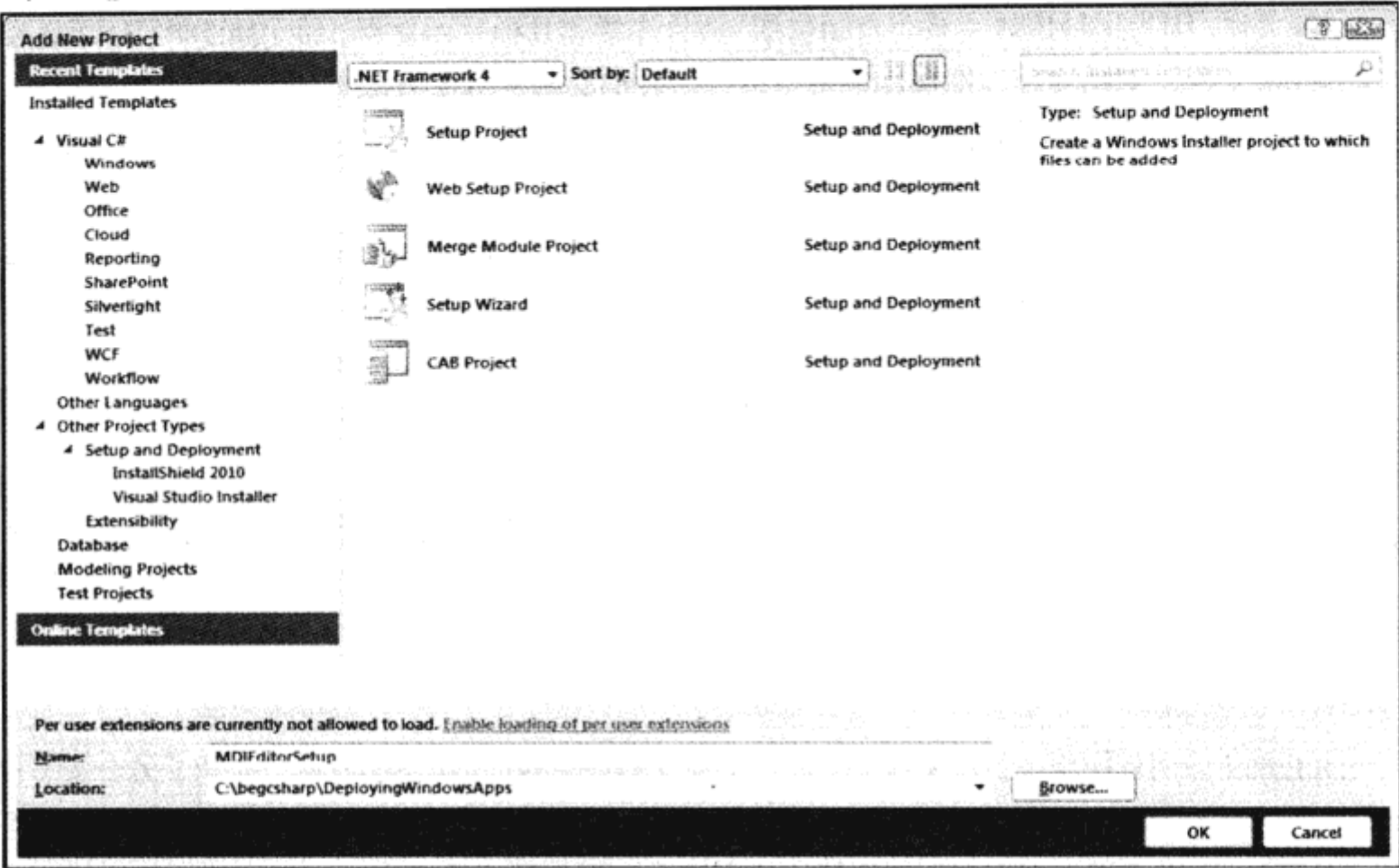


图 17-22

17.5.3 项目属性

到目前为止，我们仅仅拥有一个项目文件可以用于安装解决方案。必须定义要安装的文件，还必须配置项目属性。为此，必须了解 Packaging 和 Bootstrapper 选项的含义。

1. 打包

MSI 是启动安装的数据库，但是我们可以定义如何使用如图 17-23 所示的 3 个选项打包要安装的文件。右击 MDIEditorSetup 项目，选择 Properties，就会打开此对话框。

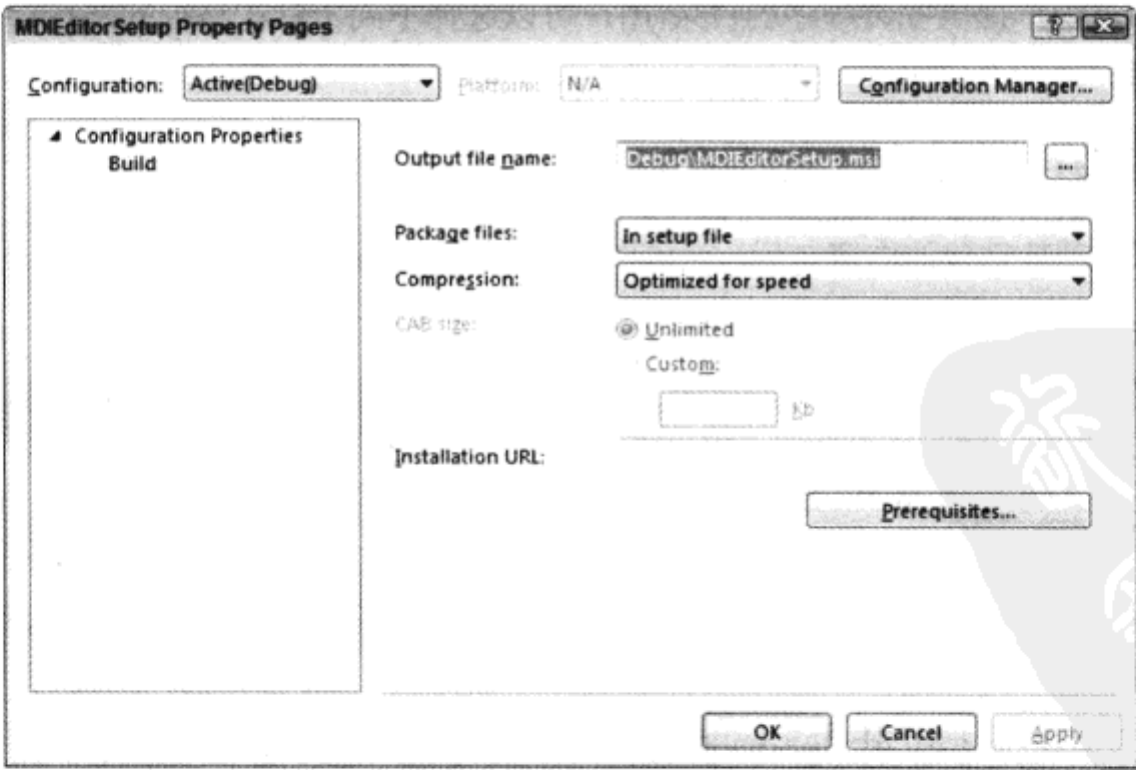


图 17-23

Package files 下拉列表中的选项如下所示:

- As loose uncompressed files 选项将所有程序和数据文件都原样存储, 不进行文件压缩。
- In setup file 选项会把所有的文件合并、压缩到 MSI 文件中。可以为软件包中的单一组件重写此选项。如果将所有文件放到一个 MSI 文件中, 就必须注意安装程序的大小要适合于希望使用的介质, 比如 CD 或软盘。如果安装的文件太多, 超过了一张软盘的容量, 可以试试在 Compression 下拉列表中选择 Optimized for size 选项, 改变压缩选项。如果其容量仍然不适合, 则可以选择下一个打包选项。
- 对文件打包的第三种方法是 In cabinet file(s)。在此方法中, MSI 文件仅用于加载和安装 CAB 文件。使用 CAB 文件可以设置文件的大小, 以便从 CD 或软盘上安装(对于从软盘的安装, 可以设置 1440KB 的安装容量)。

## 2. 预先安装的软件包

在上面的对话框中可以配置预先安装的软件包, 即在安装应用程序之前必须安装的部分。单击 Installation URL 文本框旁边的 Prerequisites 按钮, 就会弹出 Prerequisites 对话框, 如图 17-24 所示。可以看出, .NET Framework 4 Client Profile 默认选中, 作为预先安装的软件包。如果客户系统没有安装 .NET Framework, 就从安装程序中安装它。还可以选择其他预先安装选项, 如下所示。

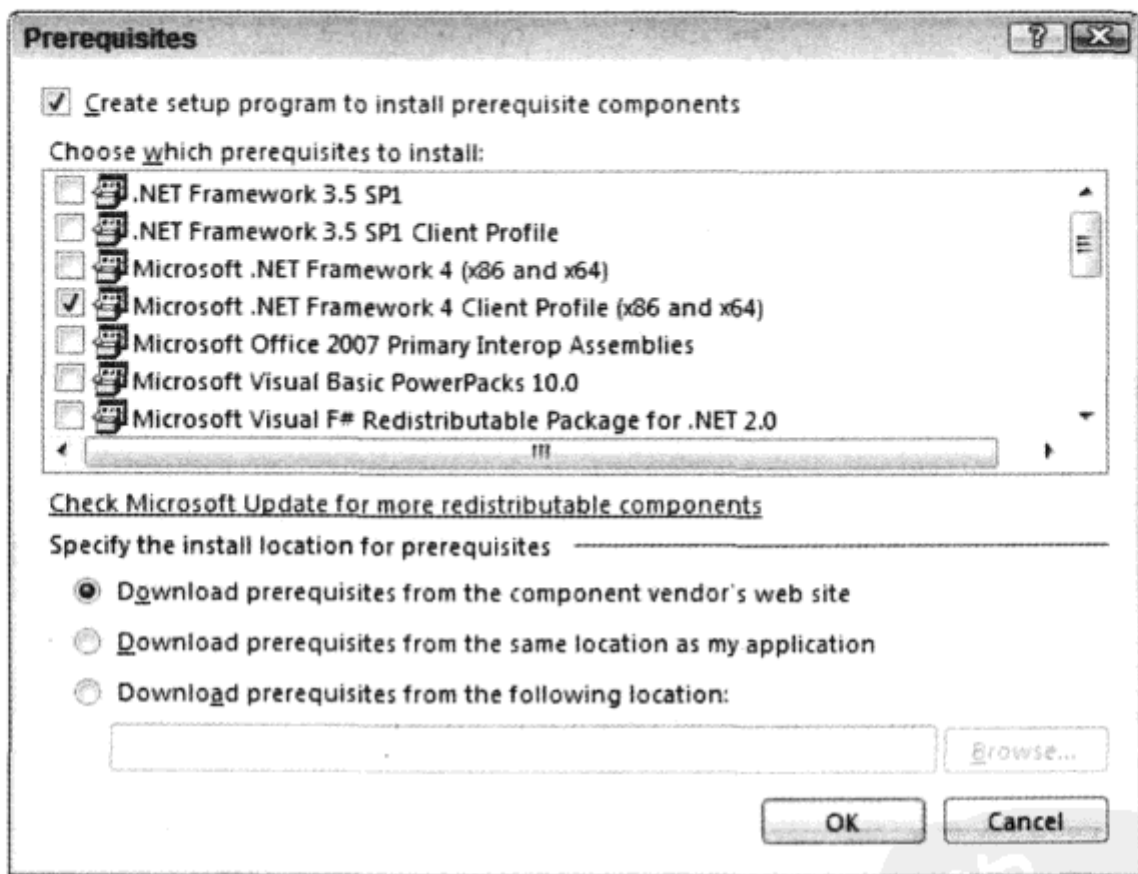


图 17-24

- Windows Installer 3.1: Windows Installer 3.1 用于通过 Visual Studio 2010 创建的安装软件包。如果目标系统是 Windows Vista 或 Windows Server 2008, 该安装程序就已经安装在系统上了。在较旧的系统上, 可能没有 Windows Installer 的正确版本, 所以应选择这个选项, 在安装程序中包含 Windows Installer 3.1。Windows 7 使用 Windows Installer 5 版本, 而 Visual Studio 2010 使用 Windows Installer 4.5 版本。

- **SQL Server 2008 Express:** 如果需要在客户系统上有一个数据库,就可以在安装程序中包含 SQL Server 2008 Express 版本。有关用 ADO.NET 访问 SQL Server 的详细内容见第 24 章。
- **Microsoft Office 2007 Primary Interop Assemblies:** 对于使用 Office 自动功能的应用程序,可以用这个组件安装 Office 2007 的主要互操作程序集。
- **Visual Basic PowerPacks 10.0:** 提供了用 Visual Basic 编程的额外特性, Visual Basic 可以用这个组件来安装。
- **Visual F# Redistributable Package:** 对于用编程语言 F#编写的应用程序,需要这个软件包。

试一试：配置项目

(1) 将 Property 页面上的 Prerequisites 选项(View | Property Pages)改为包含 Windows Installer 3.1, 这样应用程序就可以安装在没有 Windows Installer 3.1 的系统上。将输出文件名改为 WroxMDI Editor.msi, 如图 17-25 所示。然后单击 OK 按钮。

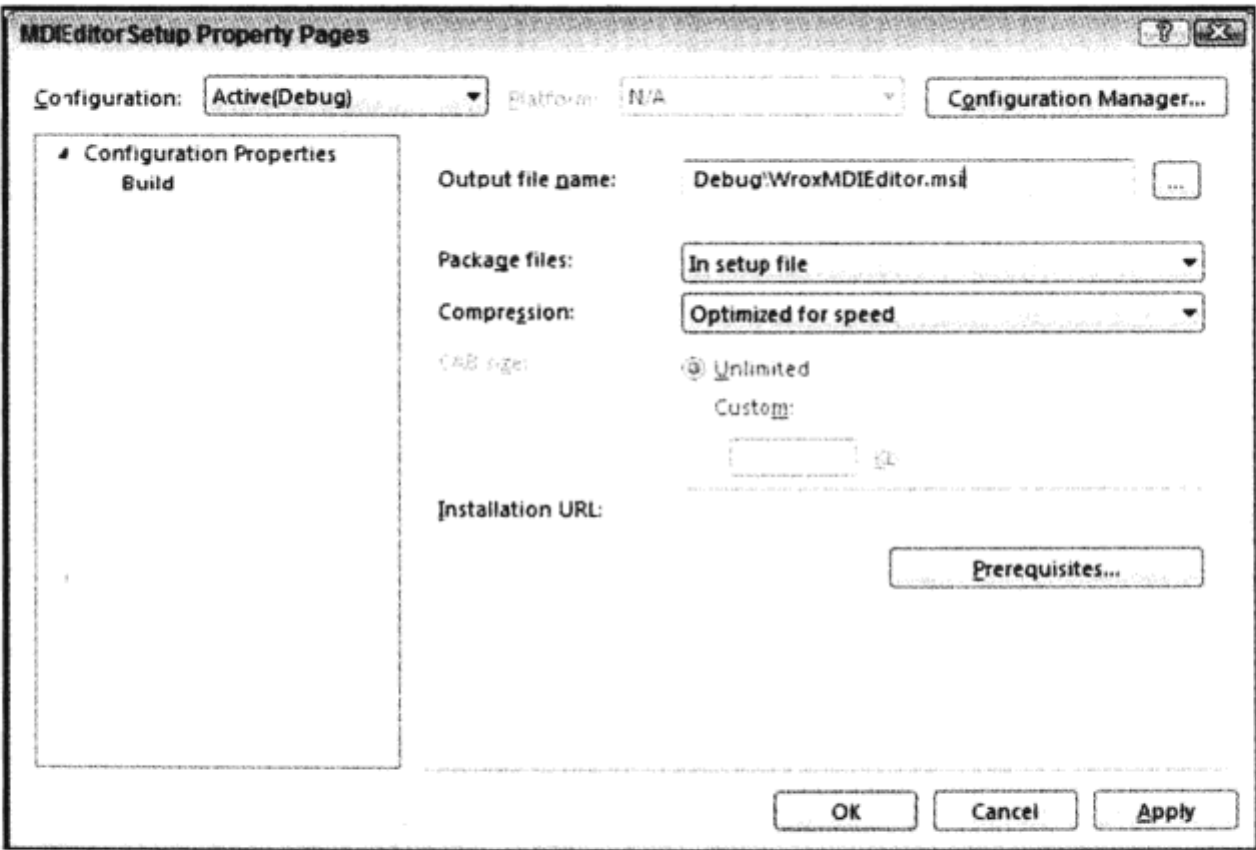


图 17-25

(2) 使用 Properties 窗口将项目属性设置为表 17-1 中的值。

表 17-1

属 性	值
Author	Wrox Press
Description	要打印和编辑文本文件的 MDI Editor
Keywords	Installer、Wrox Press、MDI Editor
InstallAllUsers	True
Manufacturer	Wrox Press
ManufacturerUrl	http://www.wrox.com

(续表)

属 性	值
Product Name	Wrox MDI Editor
SupportUrl	http://p2p.wrox.com
Title	MDI Editor 的安装演示
Version	1.0.0

17.5.4 安装编辑器

Visual Studio 2010 的 Setup Project 有 6 个编辑器。选择编辑器的方式如下：打开部署项目，选择 View | Editor 菜单项。

- File System 编辑器用于向安装软件包中添加文件。
- 使用 Registry 编辑器，可以为应用程序创建注册表项。
- File Types 编辑器允许注册应用程序的特定文件扩展名。
- 使用 User Interface 编辑器，可以添加和配置在安装程序期间显示的对话框。
- Custom Actions 编辑器允许在安装和卸载期间启动定制的程序。
- 使用 Launch Conditions 编辑器，可以指定对应用程序的要求，比如，必须已安装.NET 运行库。

17.5.5 File System 编辑器

使用 File System 编辑器，可以向安装软件包中添加文件，并配置安装它们的位置。通过 View | Editor | File System 菜单可以打开此编辑器。还会自动打开一些预定义的特殊文件夹，如图 17-26 所示。

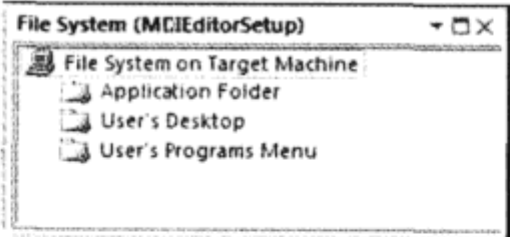


图 17-26

- Application Folder 文件夹用于存储可执行文件和库。其位置定义为[ProgramFilesFolder]\[Manufacturer]\[ProductName]。在 English 语言系统上，[ProgramFilesFolder]解析为 C:\Program Files。[Manufacturer]和[ProductName]的目录由 Manufacturer 和 ProductName 项目属性定义。
- 如果希望在桌面上放置图标，可以使用 User's Desktop 文件夹。此文件夹的默认路径是 C:\Users\username\Desktop 或 C:\Users\All Users\Desktop，这取决于安装是用于单个用户还是所有用户。
- 用户通常在 All Programs 菜单中启动程序。默认路径是 C:\Documents and Settings\username\Start Menu\Programs。在此菜单中可以把快捷方式放到应用程序上。快捷方式的名称包括公司和应用程序的名称，这样用户就很容易识别应用程序，如 Microsoft Excel。

一些应用程序创建了子菜单，由此可以启动多个应用程序，如 Microsoft Visual Studio 2010。根据 Windows 准则，许多程序由于错误的原因而列举了大量不需要的程序：在此菜单中不应该放置卸载程序，因为此功能可以在 Control Panel 的 Programs and Features 中获取，卸载程序应该在控制面



板中进行。此菜单中也不应该放置帮助文件，因为帮助文件可以直接从应用程序中获取。因此对于许多应用程序而言，在 All Programs 菜单中直接放置应用程序的快捷方式就足矣。这些限制的目的在于不要在 Start 菜单中放置太多的项，避免使它过于拥挤。

此信息的参考信息位于“Microsoft Windows 7 桌面应用程序规范”中。该文档在 <http://msdn.microsoft.com/en-us/windows/dd203105.aspx> 上。

右击并选择 Add Special Folder，还可以添加其他文件夹，其中一些文件夹如下：

- Global Assembly Cache (GAC)Folder 表示安装共享程序集的文件夹。GAC 用于在多个应用程序之间共享的程序集。
- User's Personal Data Folder 是用户的默认文件夹，其中存储了文档。C:\Users\[username] \My Documents 是默认路径。此路径是 Visual Studio 用于存储项目的默认目录。
- 当选择文件时，User's Send To Menu 中的快捷方式可以扩展 Send To 关联菜单。使用此关联菜单，用户通常可以将文件发送到目标位置，如软驱、邮件接收者或 My Documents 文件夹。

1. 向特殊文件夹添加项

选择文件夹，再选择菜单 Action | Add Special Folder，可以从列表中选择一些项，添加到特定的文件夹中。可以选择 Project Output、Folder、File 或 Assembly。给文件夹添加项目的输出，会自动添加生成的输出文件，以及一个.dll 或.exe，这取决于所添加的项目是组件库还是应用程序。选择 Project Output 或 Assembly 可以给文件夹自动添加所有的依赖关系(所有引用的程序集)。

2. 文件属性

在文件夹中选择文件的属性，就可以设置表 17-2 中所示的属性。根据文件类型，可能无法应用其中一些属性，也可能应用其中未列出的一些属性。

表 17-2

属 性	说 明
Condition	可以使用此属性定义一个条件，确定是否选择应安装的文件。如果仅在特定的操作系统版本上添加该文件，或者用户必须在对话框中进行一些选择，该属性就非常有用
Exclude	如果不安装此文件，则该属性设置为 True。这样文件就可以保留在项目中，而不会安装。如果可以肯定它不是相关文件，或它已存在于要部署应用程序的每个系统中，则可以排除该文件
PackageAs	利用 PackageAs 可以重写文件添加到安装软件包的默认方式；例如，如果项目配置是 in setup file，就可以使用此选项将特定文件的软件包配置改为 Loose，这样此文件就不会添加到 MSI 数据库文件中。如果希望添加用户在开始安装之前读取的 ReadMe 文件，这就非常有用。显然，即使压缩了其他文件，也不应压缩此文件
Permanent	将此属性设置为 True 意味着，在卸载之后文件仍会保留在目标计算机上。这可以用于配置文件。例如，安装 Microsoft Outlook 的新版本时，如果配置了 Microsoft Outlook，则在卸载并再次安装它时，就不需要再次配置它，因为上一次的安装配置并未删除
ReadOnly	此属性在安装时设置只读文件特性
Vital	此属性表示，此文件对于此产品的安装至关重要。如果此文件的安装失败，就终止完整的安装进程，只能回滚

下面的示例给 Windows 安装程序包添加应部署的文件。

### 试一试：向安装软件包添加文件

(1) 向安装程序项目的 **Application** 文件夹添加 MDI Editor 项目的主要输出，为此使用 **Project | Add | Project Output** 菜单项。在 **Add Project Output Group** 对话框中，选择 **Primary Output**，如图 17-27 所示。

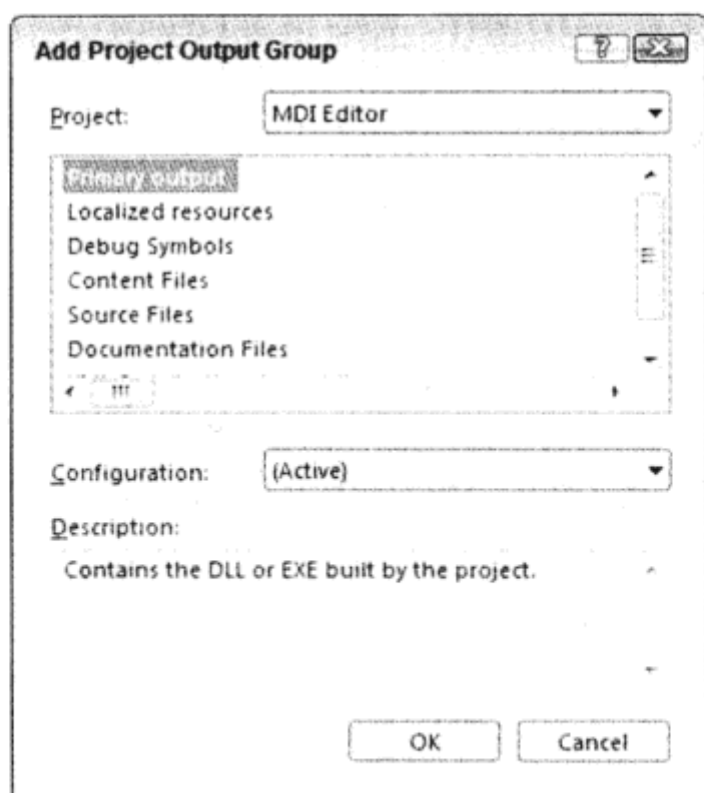


图 17-27

单击 **OK** 按钮，把 MDI Editor 项目的主要输出结果添加到自动打开的 **File System Editor** 中的 **Application Folder**。在此，主要输出就是 **MDI Editor.exe**。

(2) 要添加的附加文件有徽标、许可和 **ReadMe** 文件。在 **File System Editor** 中，选择 **Application Folder**，并选择菜单项 **Action | Add | Folder**，在 **Application Folder** 中创建名为 **Setup** 的子目录。



在 Visual Studio 中，只有选择了 **setup editor** 中的项，才能使用 **Action** 菜单。如果选择了 **Solution Explorer** 或 **Class View** 中的项，就不能使用 **Action** 菜单。

(3) 右击 **Setup** 文件夹，选择 **Add | File**，将文件 **wroxlogo.bmp**、**wroxsetuplogo.bmp**、**readme.rtf** 和 **license.rtf** 添加到 **Setup** 文件夹中。这些文件可以从本书的下载代码中得到，也可以自己创建这些文件。可以使用许可和 **ReadMe** 信息填充这些文本文件。无需改变这些文件的属性。这些文件会用在安装程序的对话框中。

位图 **wroxsetuplogo.bmp** 的大小应是 500 像素宽，70 像素高。该位图左边的 420 像素应只是一个背景图形，因为安装对话框的文本会覆盖这个区域。

(4) 将文件 **readme.txt** 添加到 **Application Folder** 中。在开始安装之前，用户应能读取这个文件。将属性 **PackageAs** 设置为 **vsdpaLoose**，这样此文件就不会压缩到 **Installer** 软件包中。还要将属性 **ReadOnly** 设置为 **True**，这样就不会改变文件。



项目现在包含两个 ReadMe 文件: readme.txt 和 readme.rtf。文件 readme.txt 可以由安装应用程序的用户在安装开始之前读取, 文件 readme.rtf 用于在安装对话框中显示一些信息。

(5) 将文件 demo.txt 拖放到 User's Desktop 文件夹中。只有在询问过用户是否希望安装之后, 才能安装此文件。因此, 将此文件的 Condition 属性设置为 CHECKBOXDEMO。CHECKBOXDEMO 是用户可以设置的条件, 其值必须大写。此条件只有设置为 TRUE, 才能安装此文件。稍后将定义一个对话框, 来设置此属性。

(6) 要从 Start | Programs 菜单中启动程序, 需要 MDI Editor 程序的一个快捷方式。

在 Application Folder 中选择 Primary output from MDI Editor 项, 并打开菜单项 Action | Create Shortcut to Primary output from MDIEditor。将所生成的快捷方式的 Name 属性设置为 Wrox MDI Editor, 并将此快捷方式拖放到 User's Programs 菜单中。

17.5.6 File Types 编辑器

如果应用程序使用了定制的文件类型, 并注册文件扩展名, 希望用户双击带有该扩展名的文件时就启动应用程序, 就可以使用 File Types 编辑器。此编辑器可以使用 View | Editor | File Types 启动, 图 17-28 显示的是添加了一个定制文件扩展名的 File Types 编辑器。

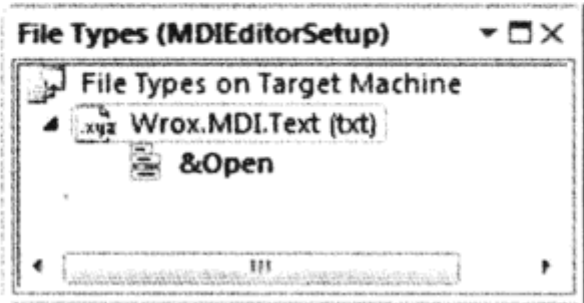


图 17-28

利用 File Type 编辑器可以配置可在应用程序中处理的文件扩展名。文件扩展名具有表 17-3 所示的属性。

表 17-3

属 性	说 明
Name	在此添加描述文件类型的有效名称。此名称显示在 File Type Editor 中, 并写入到注册表中。名称应该是唯一的。 .doc 文件类型的示例是 Word.Document.12。无须像在 Word 中那样使用 ProgID, 也可以为 .dohtml 文件扩展名使用简单的文本, 如 wordhtmlfile
Command	使用 Command 属性可以指定可执行文件, 当用户打开此类型的文件时, 就会启动这个可执行文件
Description	在此可以添加说明
Extensions	应用程序可以进行注册的文件扩展名, 在注册表中注册的文件扩展名
Icon	为文件扩展名指定要显示的图标

创建操作

在 File Type Editor 中创建了文件类型之后, 就可以添加操作(action)。自动添加的默认操作是 Open。也可以添加其他动作, 如 New 和 Print, 或者是程序可以对文件执行的任何操作。对于操作,

必须定义 Arguments 和 Verb 属性。Arguments 属性指定传递给应用程序的参数，已经为文件扩展名对其进行了注册。例如，"%1"的意思是，文件名传递给应用程序。Verb 属性指定要发生的操作。如果应用程序支持的话，对于打印操作可以添加/print。

下面给 MDIEditor 安装程序添加操作。我们希望注册文件扩展名，这样就可以在 Windows Explorer 中使用 MDIEditor 应用程序，打开带有扩展名.txt 的文件。在注册后，可以双击这些文件打开它们，自动启动 MDIEditor 应用程序。

试一试：设置文件扩展名

(1) 通过 View | Editor | File Types 菜单项启动 File Types 编辑器。使用菜单项 Action | Add File Type 添加新文件类型，其属性如表 17-4 所示。

表 17-4

属 性	值
(Name)	Wrox.MDIEditor.Text
Command	MDI Editor 的主要输出
Description	文本文档
Extensions	Txt

也可以设置 Icon 属性，为打开的文件定义图标。还可以设置 MIME 类型。  
Open 操作的属性使用默认值，这样就将文件名作为应用程序参数传递。

17.5.7 Launch Condition 编辑器

通过 Launch Condition 编辑器，可以在安装之前对目标系统指定一些要求。选择菜单项 View | Editor | Launch Conditions，启动 Launch Condition 编辑器，如图 17-29 所示。

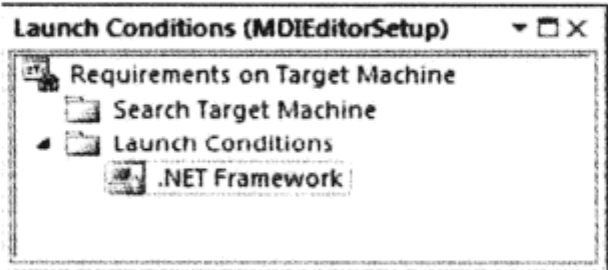


图 17-29

可以在这个编辑器的如下两部分指定要求：Search Target Machine 和 Launch Conditions。在第一部分中，可以指定要搜索什么文件或注册表项，如果搜索失败，则在第二部分中定义错误消息。下面研究一些可以使用 Action 菜单定义的启动条件。

- File Launch Condition 在开始安装前在目标系统上搜索文件。
- Registry Launch Condition 允许在安装之前查找注册表项。
- Windows Installer Launch Condition 可以搜索必须存在的 Windows Installer 组件。
- .NET Framework Launch Condition 检查目标系统是否已经安装了 .NET Framework。

- **Internet Information Services Launch Condition** 检查已经安装的 Internet Information Services。  
在 Internet Information Services 已安装的情况下，如果添加这个启动条件，会搜索定义好的特定注册表项，并添加检查特定版本的条件。

在默认情况下包含了 .NET Framework Launch Condition，其属性设置为预定义的值：message 属性设置为 [VSDNETMSG]，这是预定义的错误消息。如果未安装 .NET Framework 4，就弹出一个消息，告诉用户安装 .NET Framework。InstallUrl 默认设置为 <http://go.microsoft.com/fwlink/?linkid=131000>，这样用户就很容易地启动 .NET Framework 的安装。

### 17.5.8 User Interface 编辑器

使用 User Interface 编辑器，可以定义用户在配置安装时看到的对话框。通过它可以向用户显示许可协议，询问安装路径和配置应用程序的其他信息。

下面的示例将启动 User Interface 编辑器，以配置安装应用程序时显示的对话框。

试一试：启动 User Interface 编辑器

- (1) 选择 View | Editor | User Interface 菜单项，启动 User Interface 编辑器。
- (2) 使用 User Interface 编辑器可以为预定义的对话框设置属性。图 17-30 显示了自动生成的对话框和两个安装模式。

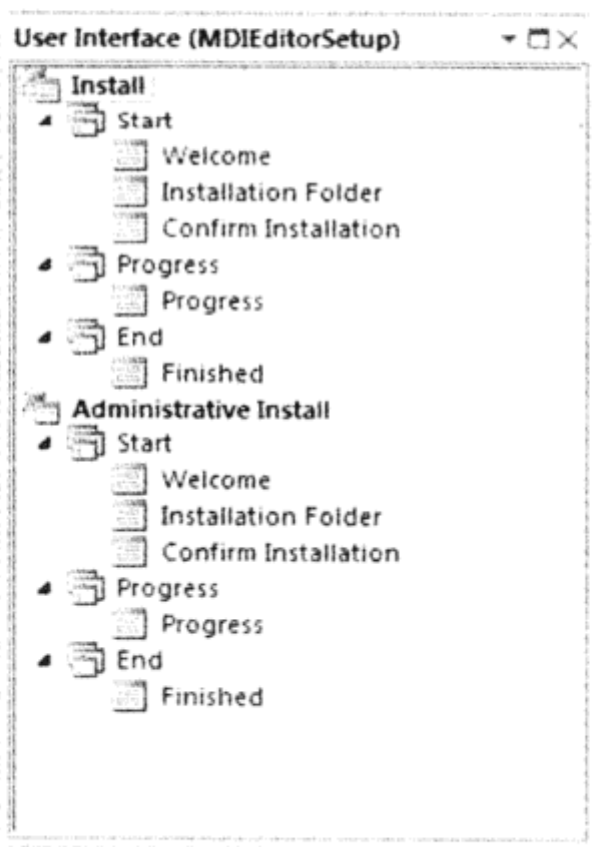


图 17-30

#### 示例的说明

如图 17-30 所示，有两种安装模式：**Install** 和 **Administrative Install**。**Install** 模式一般用于在目标系统上安装应用程序。而 **Administrative Install** 模式可以在网络共享上安装应用程序的映像，然后用户就可以从网络安装应用程序。

在两种安装模式中，显示对话框有 3 个阶段：**Start**、**Progress** 和 **End**。下面介绍默认的对话框：

- **Welcome** 对话框向用户显示欢迎消息。可以使用自己的消息代替默认的欢迎文本。用户只能取消安装或单击 **Next** 按钮。
- 通过第二个对话框 **Installation Folder**，用户可以选择安装应用程序的文件夹。如果添加了定制的对话框(稍后介绍)，就必须在此对话框之前添加。
- **Confirm Installation** 对话框是安装开始之前的最后一个对话框。
- **Progress** 对话框显示一个进度控件，以使用户查看安装进度。
- 安装结束后会显示 **Finished** 对话框。

即使没有在解决方案中打开 **User Interface** 编辑器，默认对话框也会在安装时自动显示，但是应配置这些对话框，显示对应用程序有用的消息。

下面的示例将配置在安装应用程序时显示的默认对话框。这里忽略了 **Administrative Install** 路径，只配置一般安装路径。

### 试一试：配置默认对话框

(1) 选择 **Welcome** 对话框。在属性窗口中，可以看到用于对话框的 3 个属性：**BannerBitmap**、**CopyrightWarning** 和 **WelcomeText**。选择 **BannerBitmap** 属性，方式是在组合框中单击 **Browse**，然后选择 **Application Folder\Setup** 文件夹中的文件 **wroxsetuplogo.bmp**。此文件存储的位图会显示在对话框顶部。

属性 **CopyrightWarning** 的默认文本为：

WARNING: This computer program is protected by copyright law and international treaties. Unauthorized duplication or distribution of this program, or any portion of it, may result in severe civil or criminal penalties, and will be prosecuted to the maximum extent possible under the law.(警告：此计算机程序受到版权法和国际公约保护。未经授权，复制或销售此程序，或其中的一部分，都会受到法律许可的最大限度的民事或刑事处罚。)

此文本也会显示在 **Welcome** 对话框中。如果希望发出更严厉的警告，可以改变此文本。属性 **WelcomeText** 可以定义在对话框中显示的更多文本，其默认值如下：

The installer will guide you through the steps required to install [ProductName] on your computer.(安装程序会引导您完成在计算机上安装[ProductName]的全部步骤)。

也可以改变此文本。字符串[ProductName]会自动被在项目属性中定义的属性 **ProductName** 代替。

(2) 选择对话框 **Installation Folder**。此对话框仅有两个属性 **BannerBitmap** 和 **InstallAllUsersVisible**。**InstallAllUsersVisible** 的默认值是 **true**。如果这个值设置为 **false**，就只有已登录的用户可以安装该应用程序。按照 **Welcome** 对话框的方式，把 **BannerBitmap** 属性改为 **wroxsetuplogo.bmp** 文件。每个对话框都可以用这个属性显示位图，所以也可以在其他对话框中改变此属性。

## 1. 其他对话框

如果设计了一个定制对话框，您无法将其添加到 **Visual Studio** 安装程序的安装序列中。完成这个任务需要更专业的工具，如 **Windows** 的 **InstallShield** 或 **Wise**。但是有了 **Visual Studio** 安装程序，可以在 **Add Dialog** 窗口中添加和定制许多预定义的对话框。

在 User Interface 编辑器中选择 Start 序列，再选择菜单项 Action | Add Dialog，就可以打开 Add Dialog 对话框，如图 17-31 所示。所有这些对话框都是可以配置的。

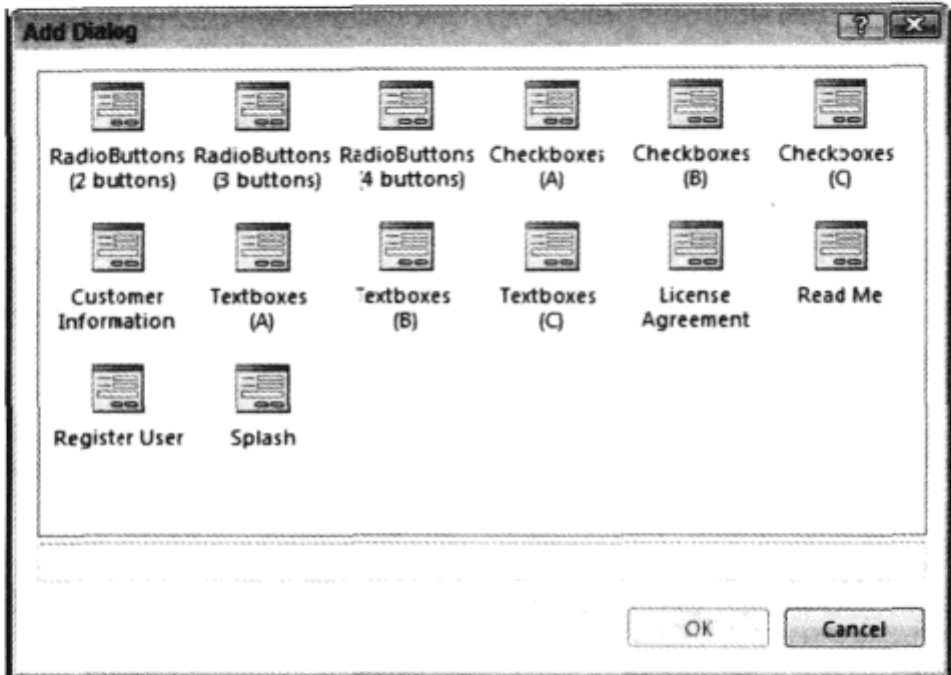


图 17-31

其中的对话框，有的可以显示 2、3 或 4 个单选按钮，复选框对话框至多可以显示 4 个复选框，文本对话框至多可以显示 4 个文本框。设置属性，就可以配置这些对话框。

下面快速浏览一下其中一些对话框：

- Customer Information 对话框询问用户的姓名、公司和产品的序列号。如果没有为产品提供序列号，可将 ShowSerialNumber 设置为 false，隐藏 Serial Number 文本框。
  - 使用 License Agreement 对话框，用户可以在开始安装之前接受一个许可。许可文件通过 LicenseFile 属性定义。
  - 用户可以在 Register User 对话框中单击 Register Now 按钮，启动使用 Executable 属性定义的一个程序。该定制程序可以向 FTP 服务器发送数据，或通过电子邮件传输数据。
  - Splash 对话框在开始安装之前显示一个闪屏，其中使用了由 SplashBitmap 属性指定的位图。
- 在下面的示例中要添加一些对话框，如 Read Me、License Agreement 和 Checkboxes 对话框。

试一试：添加其他对话框

(1) 使用 Action | Add Dialog 菜单项，可以在 Start 序列中添加 Read Me、License Agreement 和 Checkboxes(A)对话框。以拖放方式，定义启动序列中的顺序：

```
Welcome - Read Me - License Agreement - Checkboxes(A) - Installation Folder - Confirm Installation.
```

- (2) 按以前的方式为所有这些对话框配置属性 BannerBitmap。对于 Read Me 对话框，将属性 ReadmeFile 设置为 readme.rtf，即前面给 Application Folder\Setup 添加的文件。
- (3) 对于 License Agreement 对话框，将 LicenseFile 属性设置为 license.rtf。
- (4) Checkbox(A)对话框用于询问用户是否安装 User's Desktop 文件夹中的文件 demo. wroxtext。根据表 17-5 来改变此对话框的属性。

表 17-5

属 性	值
BannerText	Optional Files
BodyText	Installation of optional files
Checkbox1Label	Do you want a demo file put on to the desktop?
Checkbox1Property	CHECKBOXDEMO
Checkbox2Visible	False
Checkbox3Visible	False
Checkbox4Visible	False

Checkbox1Property 属性值设置为与文件 demo.txt 的 Condition 属性相同——前面使用 File System 编辑器向软件包中添加文件时设置了这个 Condition 属性。如果用户选中此复选框，则 CHECKBOXDEMO 的值是 true，就安装文件；如果未选中复选框，其值是 false，就不安装文件。其他复选框的 CheckboxXVisible 属性设置为 false，因为仅需要一个复选框。

### 17.6 生成项目

现在开始生成安装程序项目。

试一试：生成项目

- (1) 要创建 Windows 安装程序软件包，右击 SimpleEditorSetup 项目，选择 Build 选项。
- (2) 如果生成成功，则可以在 Debug 或 Release 目录中找到 setup.exe、WroxSimpleEditor.msi 和 readme.txt 文件(取决于生成设置)。

示例的说明

setup.exe 启动 MSI 数据库文件 WroxSimpleEditor.msi 的安装。向安装程序项目添加的所有文件(只有一个例外)都合并、压缩到 MSI 文件中，因为项目属性设置为 Package Files in Setup File。其中一个例外是文件 readme.txt。由于改变了 PackageAs 属性，该文件可以在安装应用程序之前读取。NET Framework 的安装软件包位于 DotNetFx 子目录中。

### 17.7 安装

现在可以开始安装 MDI Editor 应用程序。双击 setup.exe 文件，或选择文件 WroxMDIEditor.Msi。按下鼠标右键，打开关联菜单，选择 Install 选项。也可以在 Visual Studio 2010 中启动安装，即右击在 Solution Explorer 中打开的安装项目，选择 Install 选项。

从下面的屏幕图中可以看到，所有的对话框都具有 Wrox 标志，插入的 Read Me 和 License Agreement 对话框带有已配置的文件。



17.7.1 Welcome

第一个显示的对话框是 Welcome 对话框(如图 17-32 所示)。在此对话框中, 可以看到 Wrox 徽标, 它是通过设置 BannerBitmap 属性的值而插入的。所看到的文本由属性 WelcomeText 和 CopyrightWarning 定义。此对话框的标题是由属性 ProductName 设置的, 该属性是通过项目属性设置的。

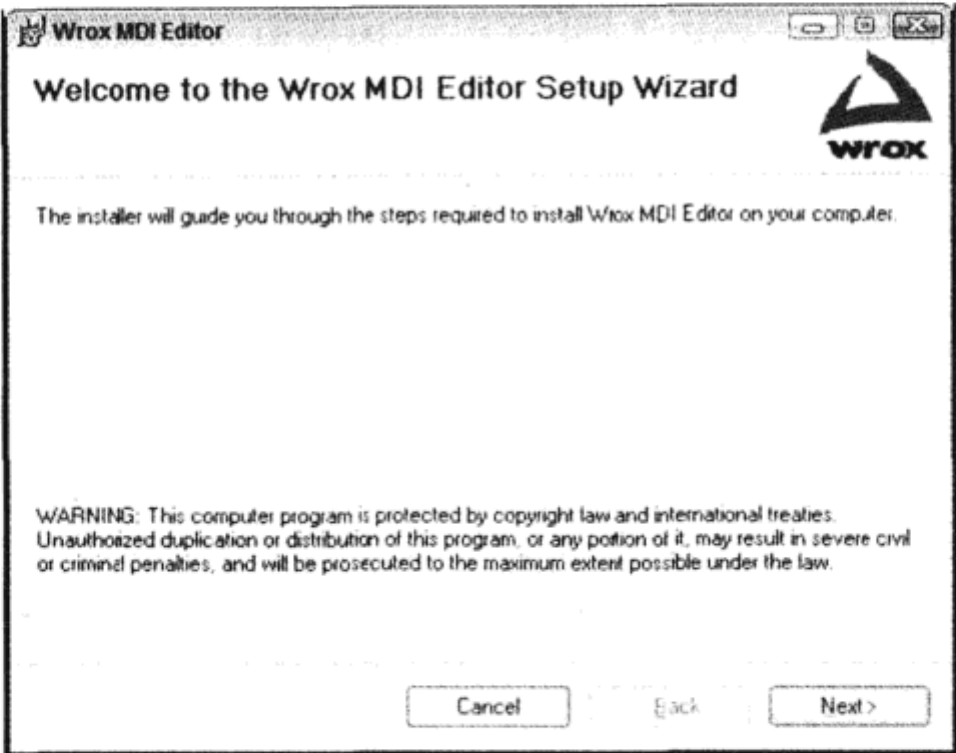


图 17-32

17.7.2 Read Me

单击 Next 按钮之后, 就会显示 Read Me 对话框。它显示通过属性 ReadmeFile 配置的格式文本文件 readme.rtf, 如图 17-33 所示。

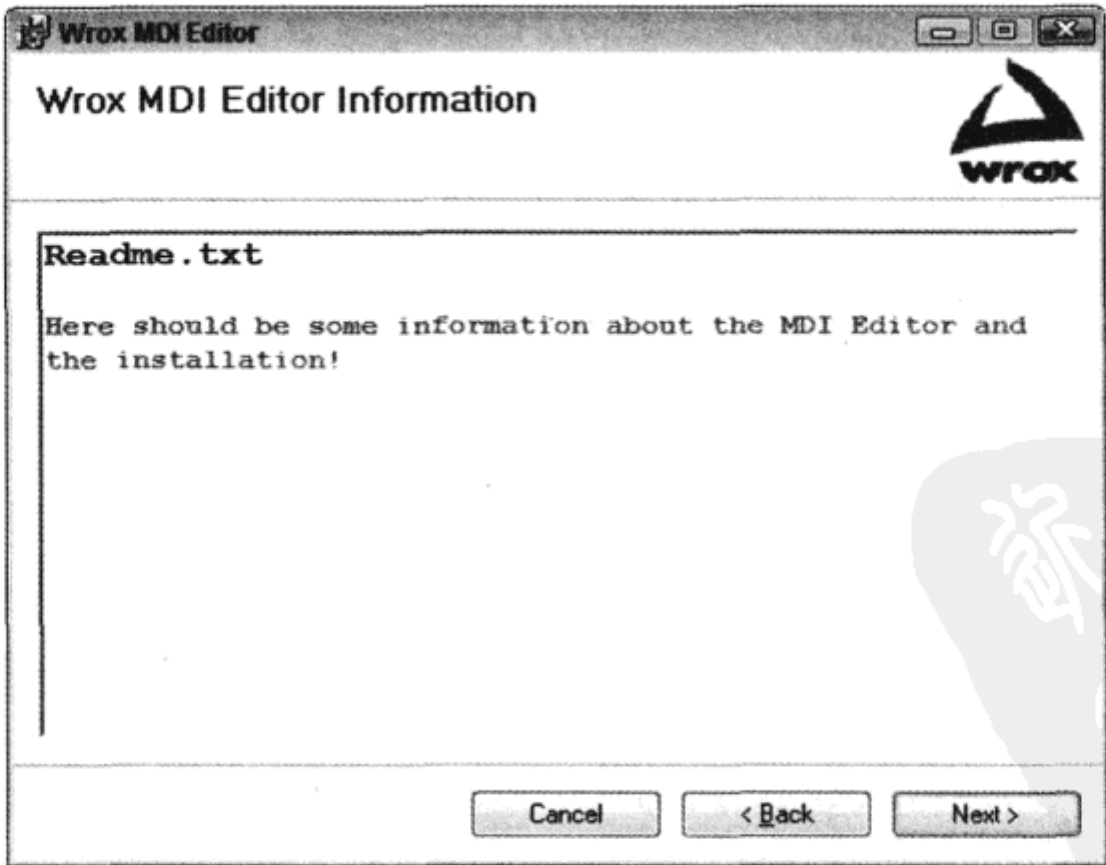


图 17-33



17.7.3 License Agreement

第 3 个对话框是 license agreement。在此仅配置了 BannerBitmap 和 LicenseFile 属性。同意此许可的单选按钮是自动添加的。如图 17-34 所示，只有选中 I Agree 按钮，才能激活 Next 按钮。这个功能是自动完成的。

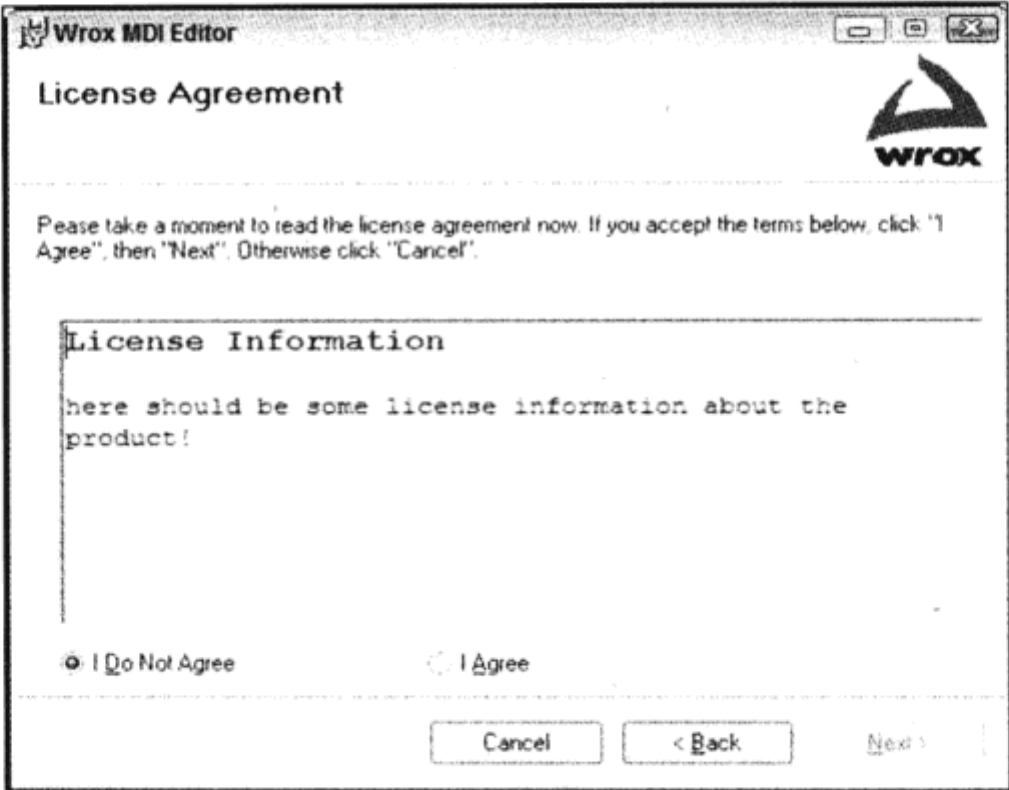


图 17-34

17.7.4 Optional Files

同意许可信息并单击 Next 按钮后，就显示 Checkboxes(A)对话框，如图 17-35 所示。在此对话框中，可以看到用属性 BannerText、BodyText 和 Checkbox1Label 定义的文本。其他复选框都不可见，因为特定 CheckboxVisible 属性设置为 false。

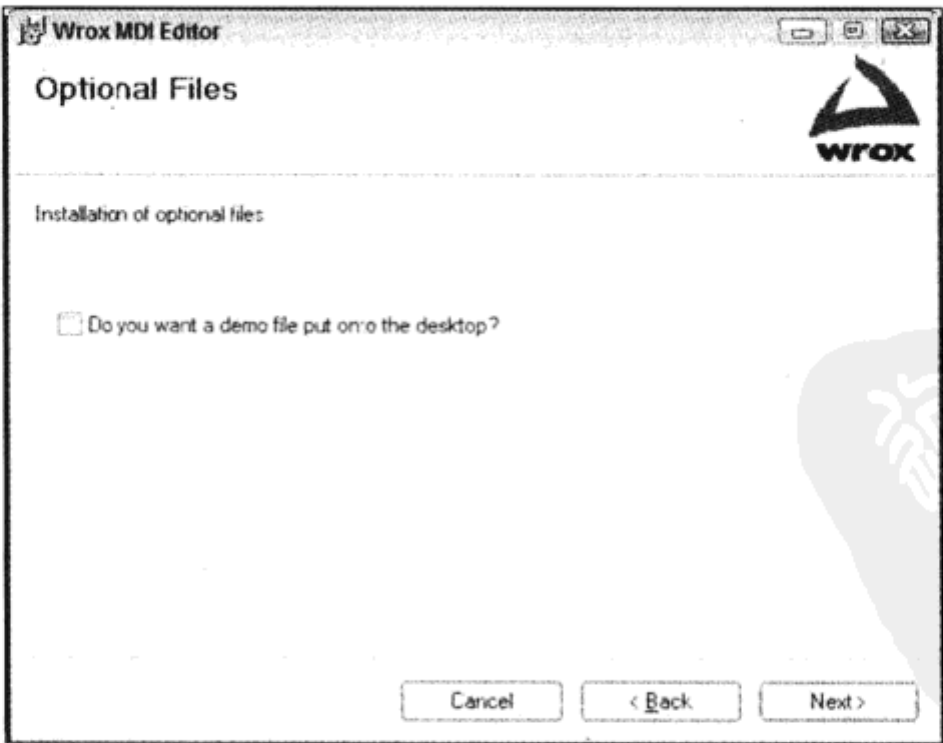


图 17-35

选中复选框就会在桌面上安装 demo.txt 文件。

17.7.5 选择安装文件夹

在 Select Installation Folder 对话框中，用户可以选择安装应用程序的路径，如图 17-36 所示。此对话框只允许设置属性 BannerBitmap。默认路径是[Program Files][manufacturer][Product Name]。

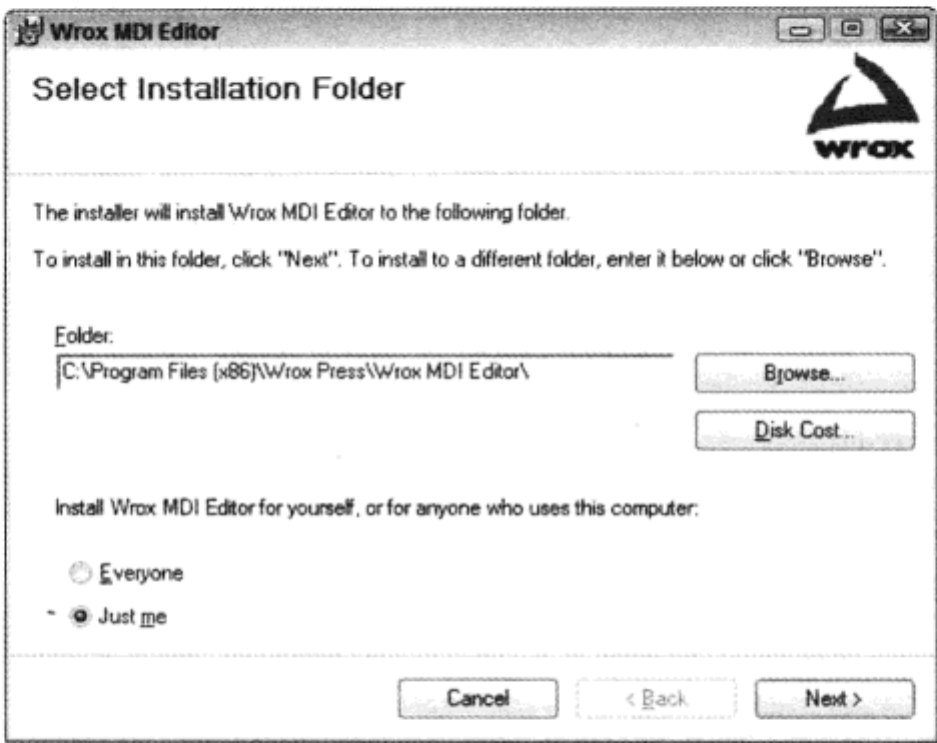


图 17-36

用户也可以指定是为所有人安装应用程序，还是只为当前登录的用户安装应用程序。根据这个选项的响应，程序文件的快捷方式会放在用户指定的目录或 All Users 目录中。

Disk Cost

单击 Disk Cost 按钮，可以打开如图 17-37 所示的对话框，其中显示了所有硬盘的磁盘空间，并计算每个磁盘所需的空

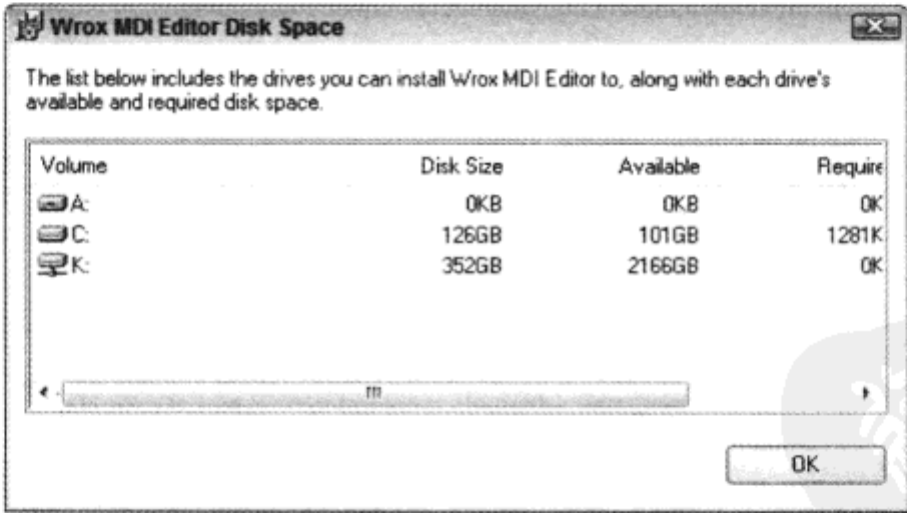


图 17-37

17.7.6 确认安装

Confirm Installation 对话框是开始安装前显示的最后一个对话框。在此不向用户询问什么问题，它仅仅是实际安装之前取消安装的最后一次机会，如图 17-38 所示。

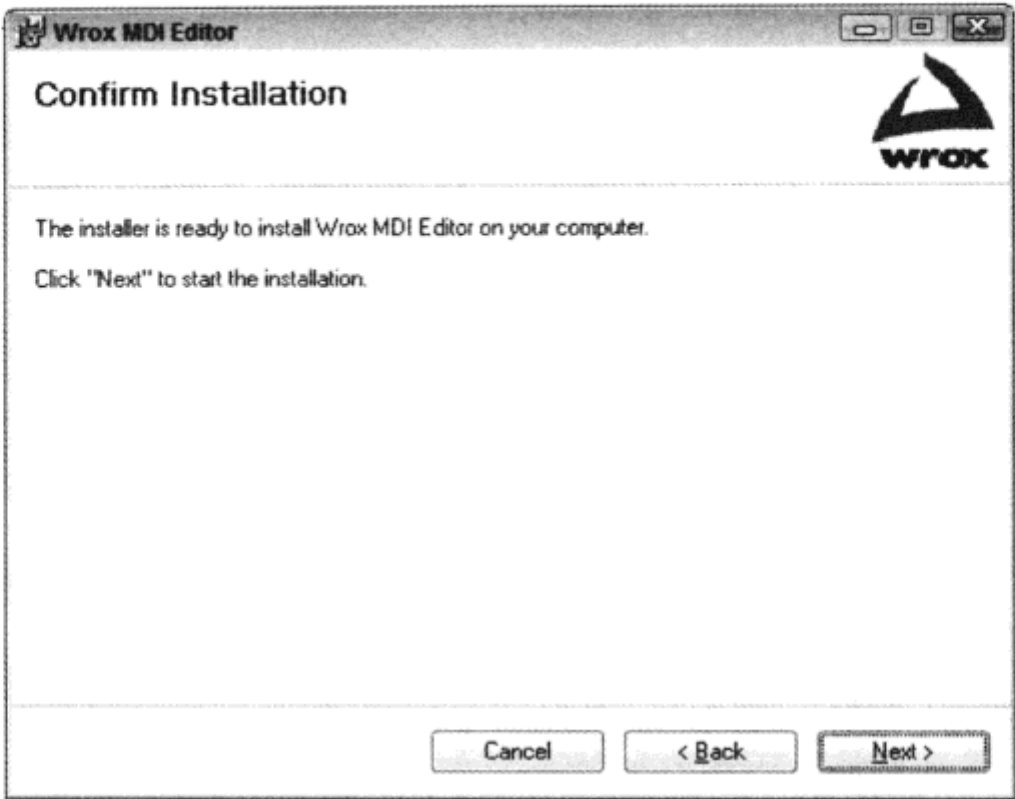


图 17-38

17.7.7 进度

Installing 对话框显示安装期间的进度控件，告诉用户安装还在进行，并大致估算安装所需要的时间。因为 MDI Editor 是一个小程序，所以此对话框很快便会结束，如图 17-39 所示。

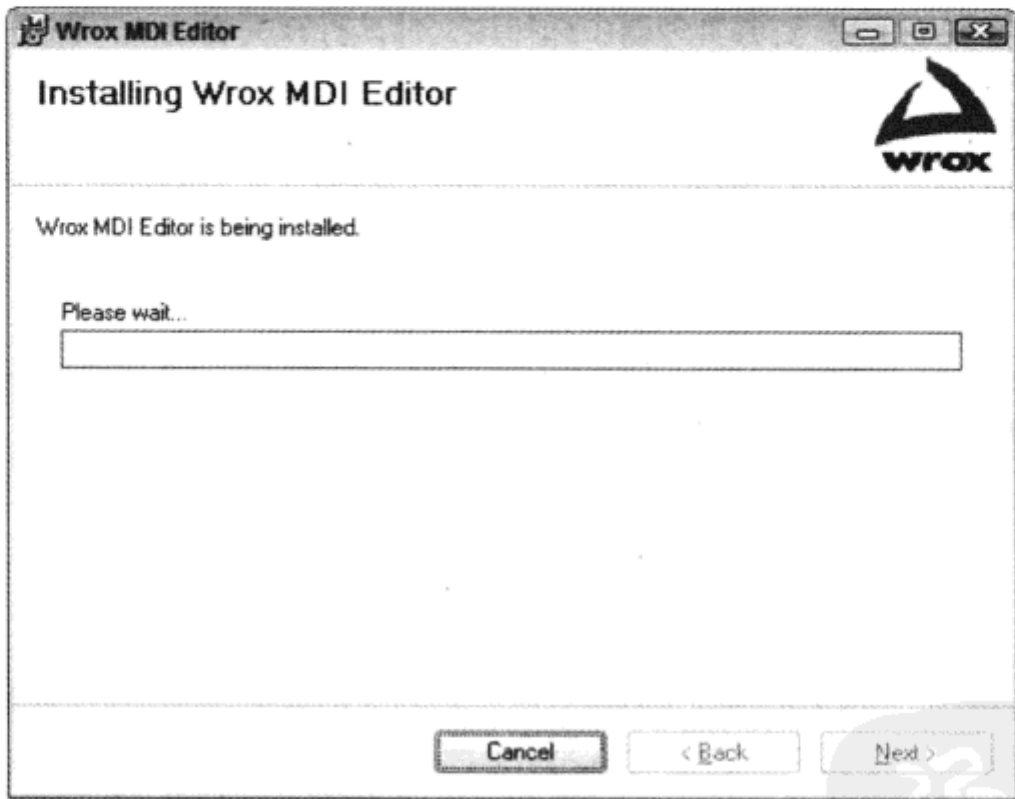


图 17-39

17.7.8 完成安装

安装成功后，会显示最后一个对话框 Installation Complete，如图 17-40 所示。

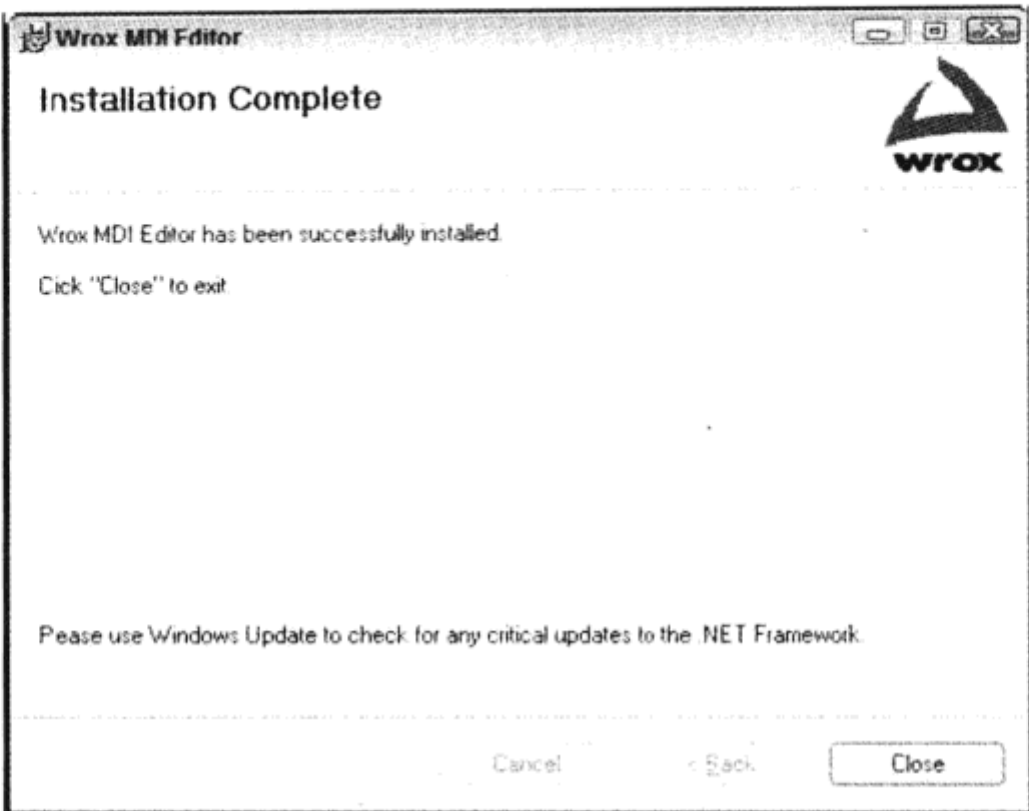


图 17-40

17.7.9 运行应用程序

选择 **Start | All Programs | Wrox MDI Editor** 菜单项可以启动这个编辑器。我们注册了文件扩展名，所以可以通过另一种方式启动应用程序：双击带有文件扩展名.txt 的文件。要选择用于打开文件的应用程序，应在 Windows 资源管理器中选择该文件，打开关联菜单。在关联菜单中选择 **Open with**，再选择需要的应用程序。要定义通过双击打开文件的应用程序，应打开关联菜单，选择 **Open with**，再选择 **Choose default program**，之后就会使用这里选择的应用程序。

如果选中了 **Optional Files** 对话框中的复选框，就可以在桌面上找到 demo.txt。

17.7.10 卸载

如果希望清除 Wrox MDI Editor，可以使用控制面板中的 **Add / Remove Programs** 命令，并在 Wrox MDI Editor 中单击 **Remove** 按钮。

17.8 小结

本章介绍了 ClickOnce 部署的用法和 Windows Installer 的功能，以及如何使用 Visual Studio 2010 创建 Installer 软件包。Windows Installer 使标准化的安装、卸载和修复变得更易于完成。

ClickOnce 是一种新技术，它使得安装 Windows 应用程序更容易，且不需要以系统管理员的身份进行登录。ClickOnce 提供了简单的部署和客户应用程序更新方式。

如果需要的功能比 ClickOnce 能提供的还多，就应使用 Windows 安装程序。Visual Studio 2010 安装程序在功能上受到了限制，不能提供 Windows 安装程序的全部功能，但是对于许多应用程序而言，Visual Studio 2010 安装程序提供的功能已经足够了。有几个编辑器可以配置生成的 Windows 安装程序文件。使用 File System 编辑器可以指定所有文件和快捷方式，Launch Conditions 编辑器可以定义一些必须预先安装的软件包，File Types 编辑器用于为应用程序注册文件扩展名，User Interface

编辑器允许更加方便地修改用于安装的对话框。  
附录 A 给出了练习答案。

17.9 练习

- (1) ClickOnce 部署有什么优点？
- (2) 用 ClickOnce 清单能定义什么？
- (3) 何时需要使用 Windows Installer？
- (4) 在 Visual Studio 中，可以使用哪些编辑器创建 Windows 安装软件包？

17.10 本章要点

主 题	重 要 概 念
ClickOnce	ClickOnce 可用于在没有管理权限的情况下部署应用程序。只需单击 Web 页面上的一个链接，就可以安装 Windows 窗体或 WPF 应用程序。这是 ClickOnce 的主要优点，因为它没有给 IT 管理员增加什么负担。可以在项目属性的 Publish 部分创建 ClickOnce 部署
Windows 安装程序包	Windows 安装程序允许在系统上安装共享的应用程序。使用这个技术，可以安装需要管理权限的应用程序组件。安装程序包很容易用 Visual Studio Installer 模板 Setup Project 创建
定制安装对话框	Visual Studio Setup Project 提供了一些预定义的对话框，可通过设置属性来定制安装，例如在安装过程中显示的版权文本和徽标



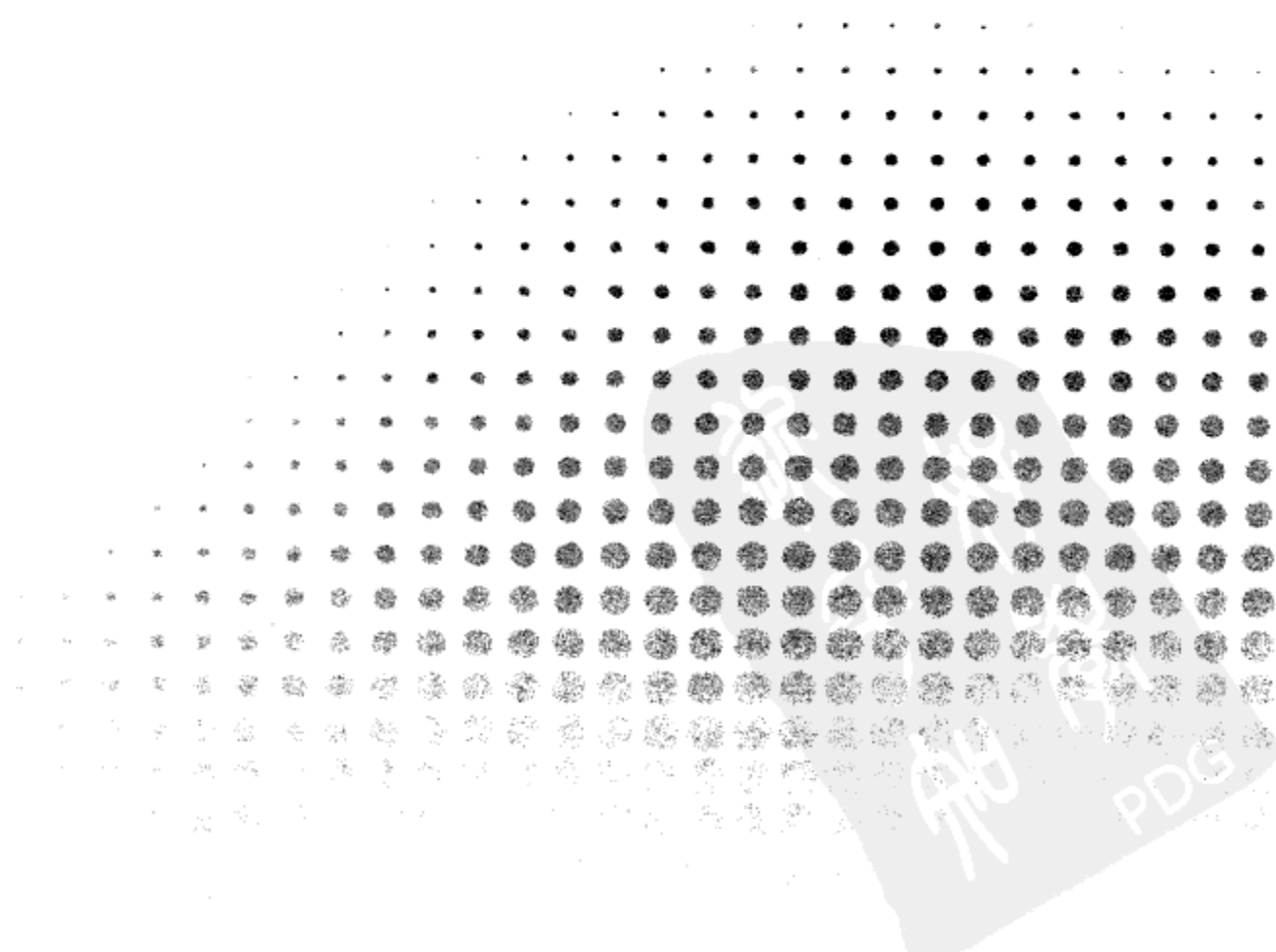


# 第Ⅲ部分

## Web 编程

---

- 第 18 章 ASP.NET Web 编程
- 第 19 章 Web 服务
- 第 20 章 部署 Web 应用程序







# 第 18 章

## ASP.NET Web 编程

### 本章内容:

---

- ASP.NET 开发概述
- 如何使用 ASP.NET 服务器控件
- 如何把 ASP.NET 回送信息发送给不同页面
- 如何创建 ASP.NET Ajax 回送
- 如何验证用户的输入
- 如何管理状态
- 如何给 Web 页面添加样式
- 如何使用母版页
- 如何实现页面的导航
- 如何验证用户的身份, 以及给用户授权
- 如何读写 SQL Server 数据库

Windows 窗体是编写 Windows 应用程序的技术, 而使用 ASP.NET 可以创建能在任意浏览器上显示的 Web 应用程序。使用 ASP.NET 编写 Web 应用程序的方式类似于开发 Windows 应用程序的方式, 这是因为服务器端控件抽象出了 HTML 代码, 并模仿了 Windows 控件的行为。当然, Windows 和 Web 应用程序仍有许多区别, 因为 Web 应用程序的底层技术是 HTTP 和 HTML。

本章将概述如何用 ASP.NET 编写 Web 应用程序, 如何使用 Web 控件、如何进行状态管理(这与 Windows 应用程序大相径庭)、如何进行身份验证以及如何在数据库中读写数据。

### 18.1 Web 应用程序概述

Web 应用程序会让 Web 服务器给客户机发送 HTML 代码。这些代码显示在 Web 浏览器, 例如 Internet Explorer 中。当用户在浏览器中输入 URL 字符串时, 就会把 HTTP 请求发送给 Web 服务器。HTTP 请求包含所请求的文件名和其他信息, 例如标识客户应用程序的字符串、客户机支持的语言

和请求所属的其他数据。Web 服务器会返回一个 HTTP 响应，其中包含 HTML 代码。Web 浏览器将解释这些 HTML 代码，向用户显示文本框、按钮和列表。

ASP.NET 技术可用于动态创建带有服务器端代码的 Web 页面。这些 Web 页面在开发时有许多地方都类似于客户端 Windows 程序。不是直接处理 HTTP 请求和响应并手工创建发送给客户端的 HTML 代码，而是使用 TextBox、Label、ComboBox 和 Calendar 等控件创建 HTML 代码。

## 18.2 ASP.NET 运行库

使用 ASP.NET 在客户系统上创建 Web 应用程序，只需一个简单的 Web 浏览器。可以使用 Internet Explorer、Opera、Netscape Navigator、Firefox 或其他支持 HTML 的 Web 浏览器。客户系统不需要安装 .NET。

在服务器系统上，需要 ASP.NET 运行库。如果系统上有 Internet Information Services (IIS)，就会在安装 .NET Framework 时为服务器配置 ASP.NET 运行库。在开发过程中，不需要 IIS，因为 Visual Studio 发布了自己的 ASP.NET Web Development Server，可以用于测试和调试 Web 应用程序。

下面讨论浏览器上的典型 Web 请求，以说明 ASP.NET 运行库的工作原理，如图 18-1 所示。客户机向服务器请求一个文件，如 default.aspx。所有的 ASP.NET Web 页面通常带有扩展名.aspx。因为这个文件扩展名是用 IIS 注册的，或者 ASP.NET Web Development Server 能识别它，所以 ASP.NET 运行库和 ASP.NET 辅助进程(worker process)就会开始工作。对文件 default.aspx 的第一次请求会启动 ASP.NET 分析器，编译器会把该文件和一个与.aspx 文件相关的 C#文件一起编译，创建一个程序集。然后 .NET 运行库的 JIT 编译器把程序集编译为本机代码。该程序集包含一个 Page 类，调用它会把 HTML 代码返回给客户端。之后删除 Page 对象。但是，会保留程序集，用于以后的请求，所以在第二个请求中，不需要再次编译程序集。

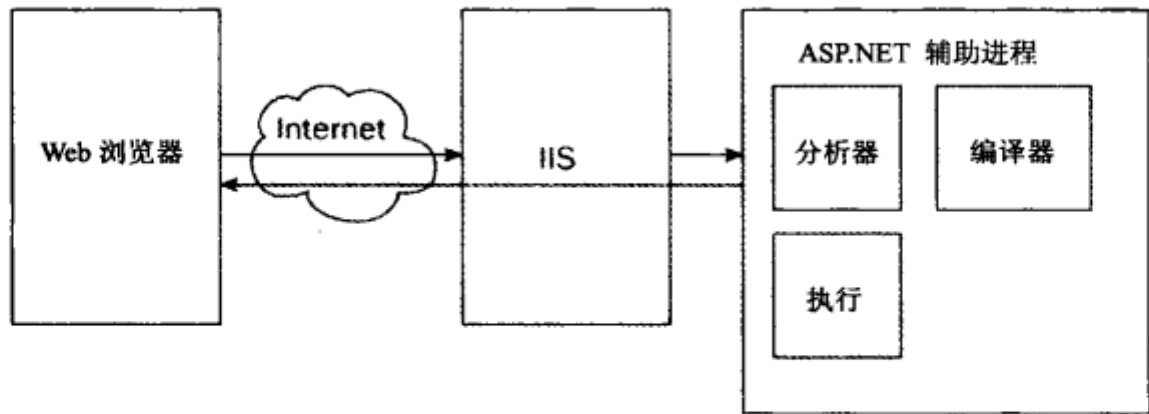


图 18-1

## 18.3 创建简单的 Web 页面

下面的示例将创建一个简单的 Web 页面。在本章和第 19 章介绍的示例应用程序中，将创建一个简单的 Event 网站，网友可以在这个网站上注册事件。

**试一试：创建一个简单的 Web 页面**

- (1) 在 Visual Studio 中选择 File | New | Project，创建一个新的 Web 项目，如图 18-2 所示。在 New

Project 对话框中, 选择 Visual C#类别, 再选择子类别 Web, 最后选择 ASP.NET Empty Web Application 模板, 将项目命名为 EventRegistrationWeb。

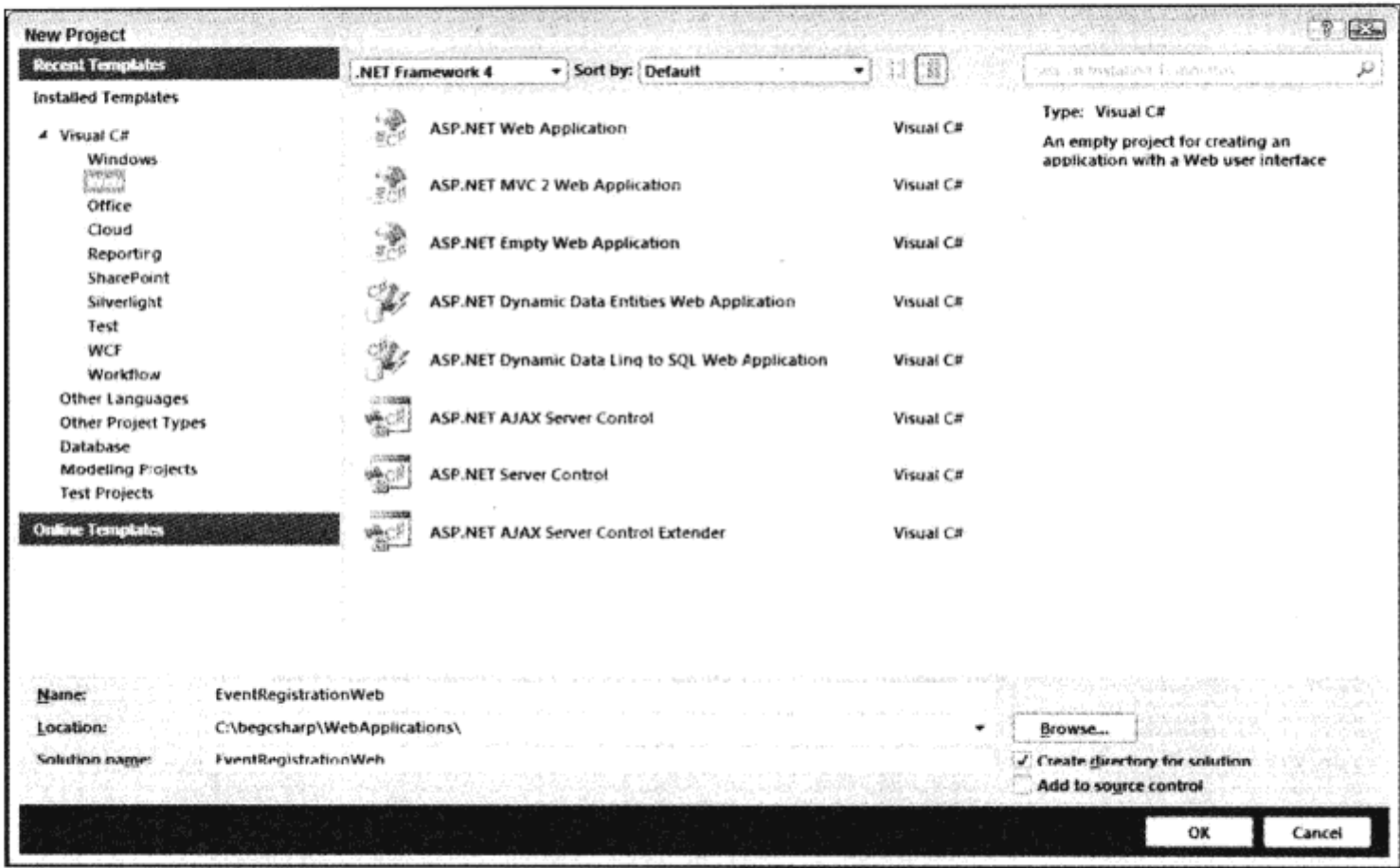


图 18-2

(2) 创建 Web 项目后, 就使用菜单 Project | Add New Item, 选择 Web Form 模板, 创建一个新的 Web 页面, 命名为 Registration.aspx, 如图 18-3 所示。

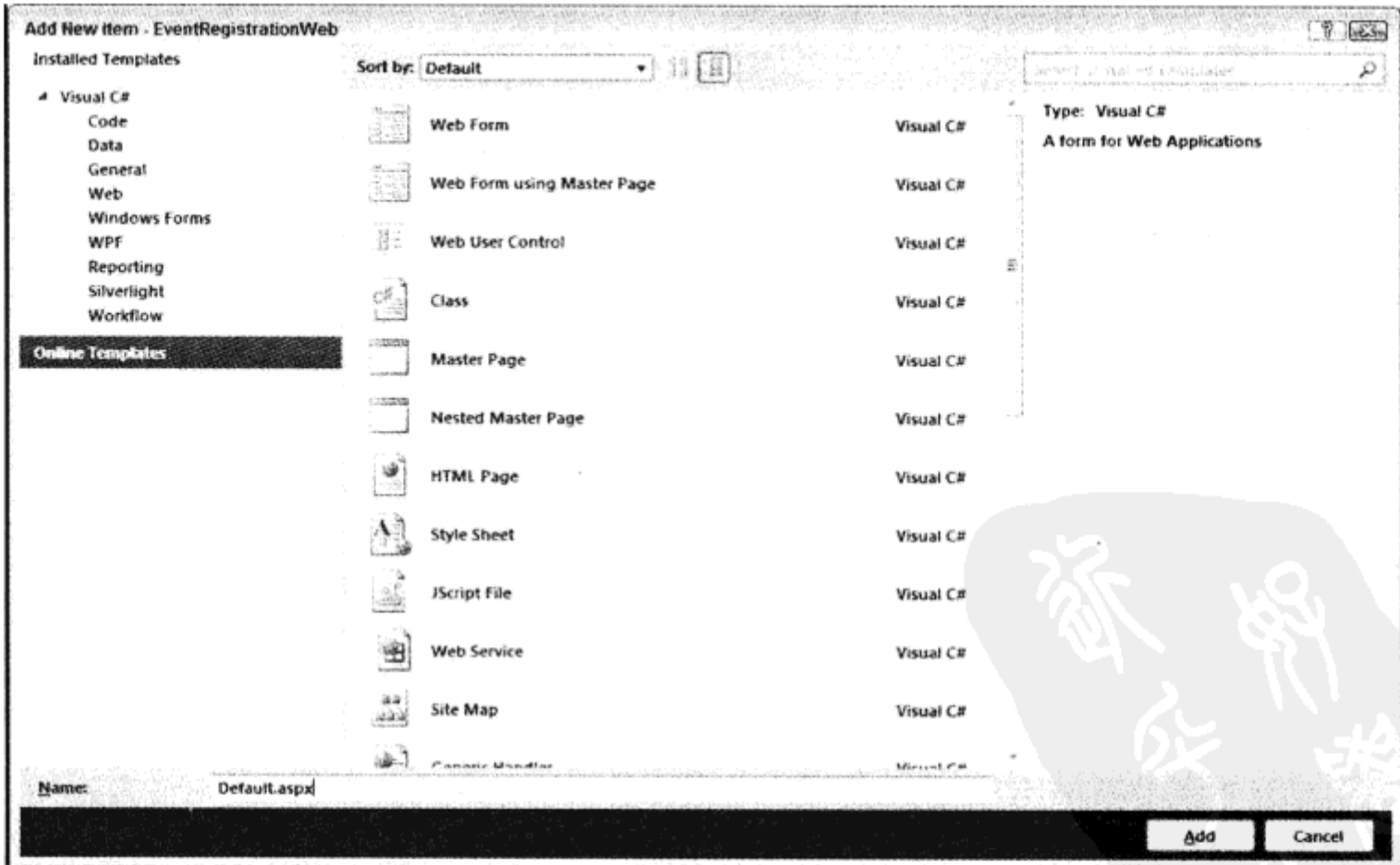


图 18-3

(3) 把控件放在一个表格中是比较有效的。单击进入设计视图，选择 Table | Insert Table 菜单，添加一个表格。在 Insert Table 对话框中，设置 5 行 2 列，如图 18-4 所示。

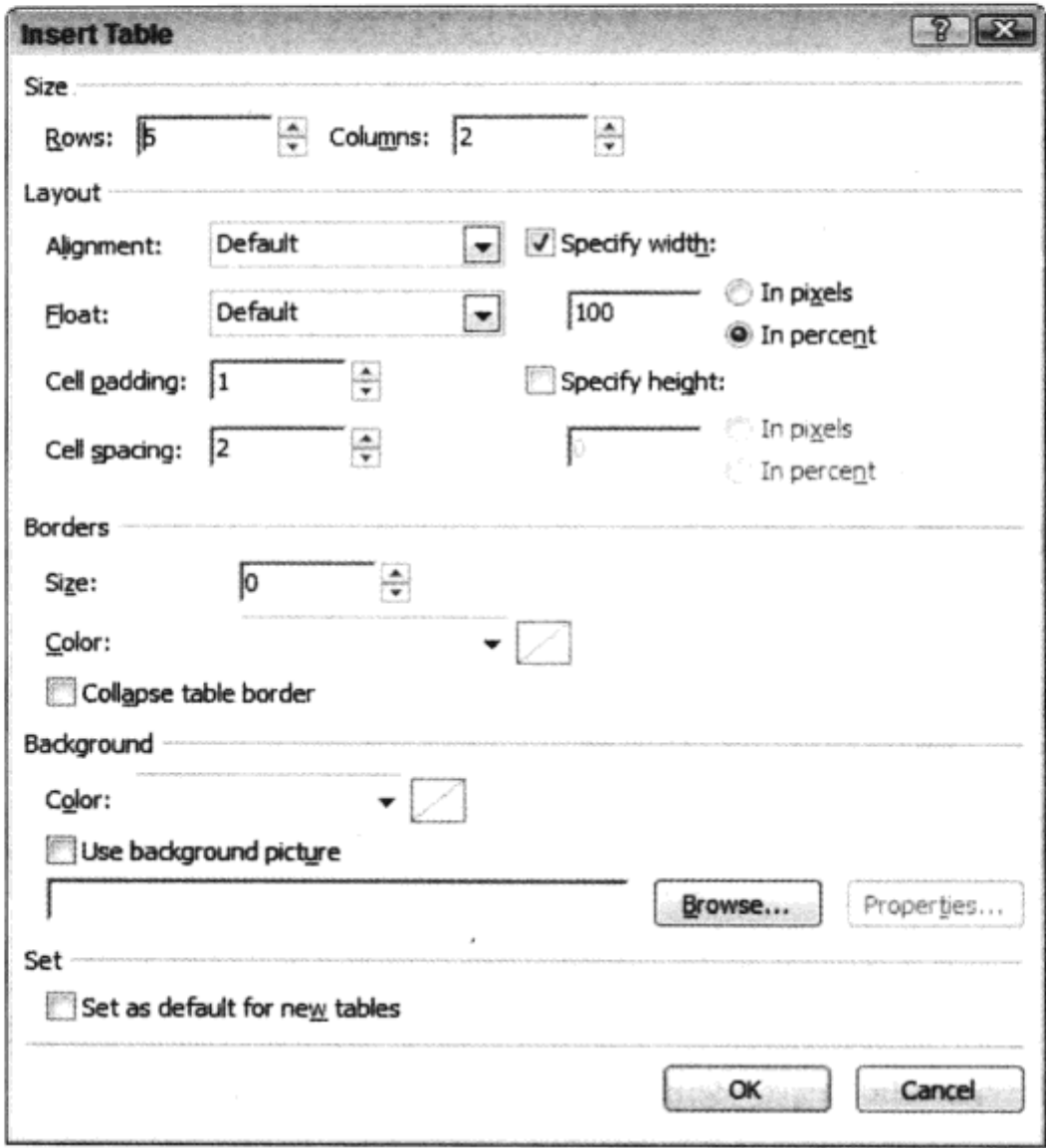


图 18-4

(4) 在表格中添加 4 个标签、3 个文本框、1 个下拉列表和 1 个按钮，如图 18-5 所示。

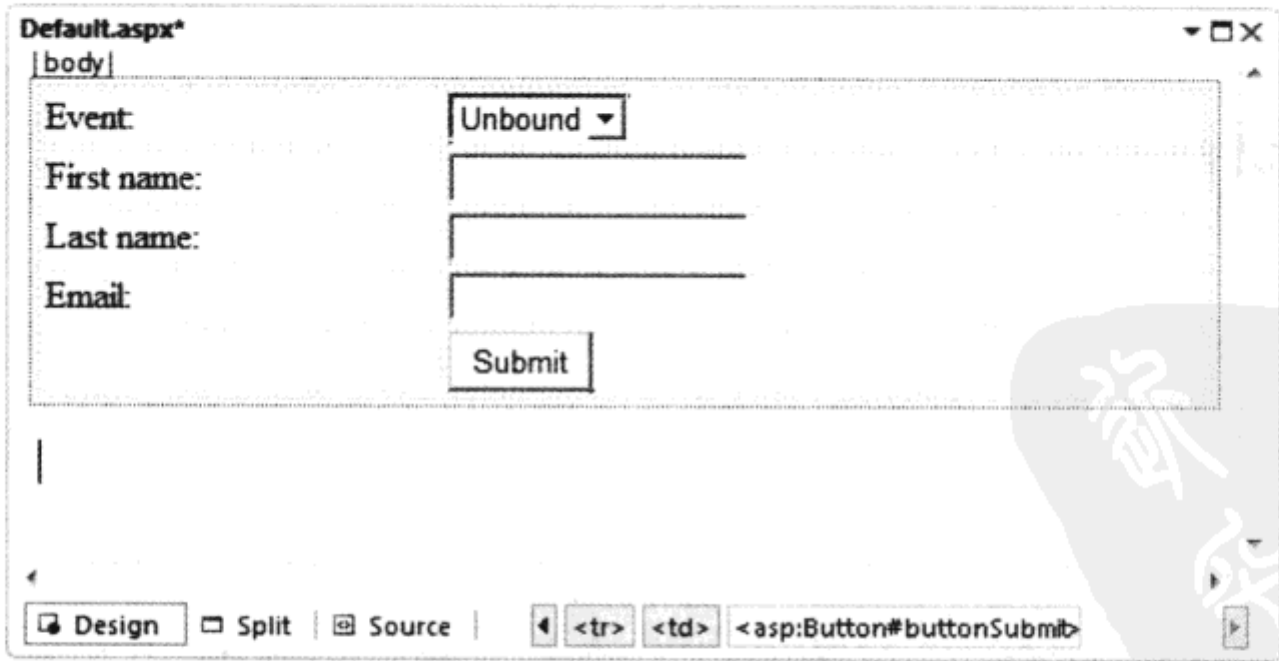


图 18-5

(5) 设置控件属性，如表 18-1 所示。

表 18-1

控 件 类 型	(ID)	文 本
Label	labelEvent	Event:
Label	labelFirstname	Firstname:
Label	labelLastname	Lastname:
Label	labelEmail	Email:
DropDownList	dropDownListEvents	
TextBox	textFirstName	
TextBox	textLastName	
TextBox	textEmail	
Button	buttonSubmit	Submit

(6) 在下拉列表中，选择 Properties 窗口中的 Items 属性，在 ListItem Collection Editor 中输入字符串 Introduction to ASP.NET、Introduction to Windows Azure 和 Take off to .NET 4.0，如图 18-6 所示。

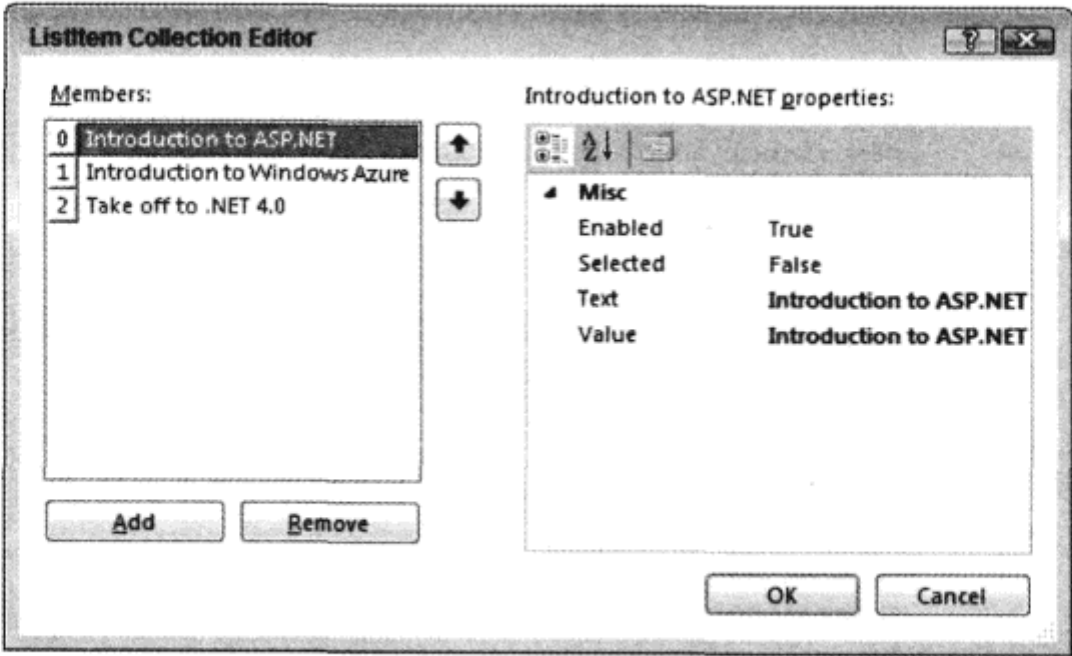


图 18-6

(7) 把编辑器切换到源代码视图，生成的代码如下所示：



可从  
wrox.com  
下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Registration.aspx.cs"
    Inherits="EventRegistrationWeb.Registration" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <style type="text/css">
        .style1
        {
```

```
        width: 100%;
    }
</style>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <table class="style1">
                <tr>
                    <td>
                        <asp:Label ID="labelEvent" runat="server" Text="Event:">
                        </asp:Label>
                    </td>
                    <td>
                        <asp:DropDownList ID="dropDownListEvents" runat="server">
                            <asp:ListItem>Introduction to ASP.NET</asp:ListItem>
                            <asp:ListItem>Introduction to Windows Azure</asp:ListItem>
                            <asp:ListItem>Take off to .NET 4.0</asp:ListItem>
                        </asp:DropDownList>
                    </td>
                </tr>
                <tr>
                    <td>
                        <asp:Label ID="labelFirstName" runat="server"
                            Text="First name:"></asp:Label>
                    </td>
                    <td>
                        <asp:TextBox ID="textFirstName" runat="server"></asp:TextBox>
                    </td>
                </tr>
                <tr>
                    <td>
                        <asp:Label ID="labelLastName" runat="server" Text="Last name:">
                        </asp:Label>
                    </td>
                    <td>
                        <asp:TextBox ID="textLastName" runat="server"></asp:TextBox>
                    </td>
                </tr>
                <tr>
                    <td>
                        <asp:Label ID="labelEmail" runat="server" Text="Email:">
                        </asp:Label>
                    </td>
                    <td>
                        <asp:TextBox ID="textEmail" runat="server"></asp:TextBox>
                    </td>
                </tr>
                <tr>
                    <td>

```



```

        &nbsp;&nbsp;&nbsp;</td>
    <td>
        <asp:Button ID="buttonSubmit" runat="server" Text="Submit" />
    </td>
</tr>
</table>
</div>
</form>
</body>
</html>

```

代码段 Registration.aspx

(8) 在启动应用程序之前，先进入项目属性，打开 Web 设置，如图 18-7 所示。验证 Start Action 设置为当前页面，在 Servers 组，验证配置了 Visual Studio Development Server。

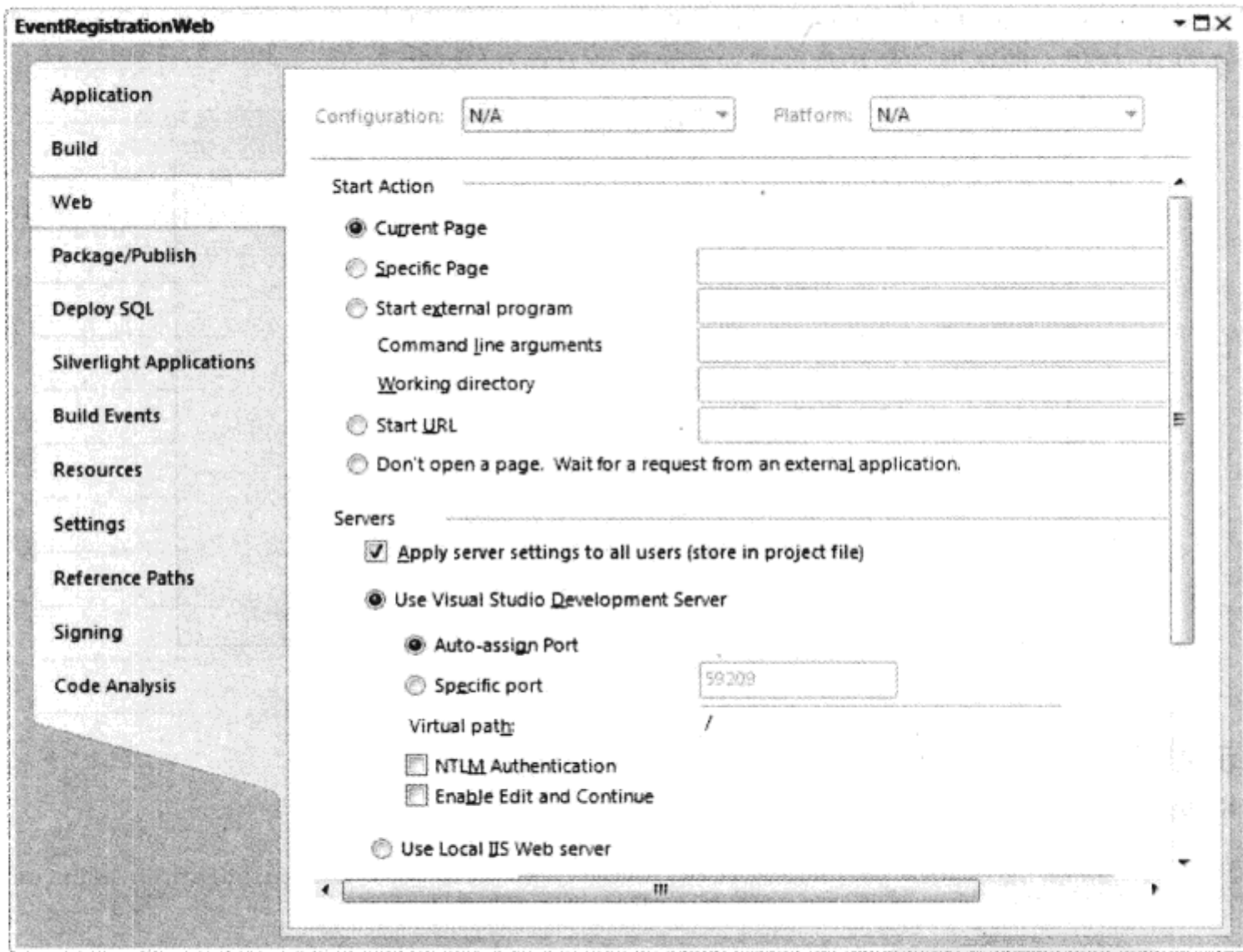


图 18-7

(9) 在编辑器中再次打开 Registration.aspx 文件。选择 Debug | Start Without Debugging 选项，启动 Web 应用程序。此时，会自动启动 ASP.NET Development Server。Windows Explorer 任务栏中有 ASP.NET Development Server 的图标。双击这个图标，就会打开如图 18-8 所示的对话框。该对话框显示了 Web 服务器的物理路径和虚拟路径，以及 Web 服务器监听的端口。该对话框还可以用于停止 Web 服务器。



图 18-8

启动应用程序，Internet Explorer 会显示 Web 页面，如图 18-9 所示。选择 View | Source 就可以查看 HTML 代码。服务器端的控件此时已转换为纯 HTML 代码。

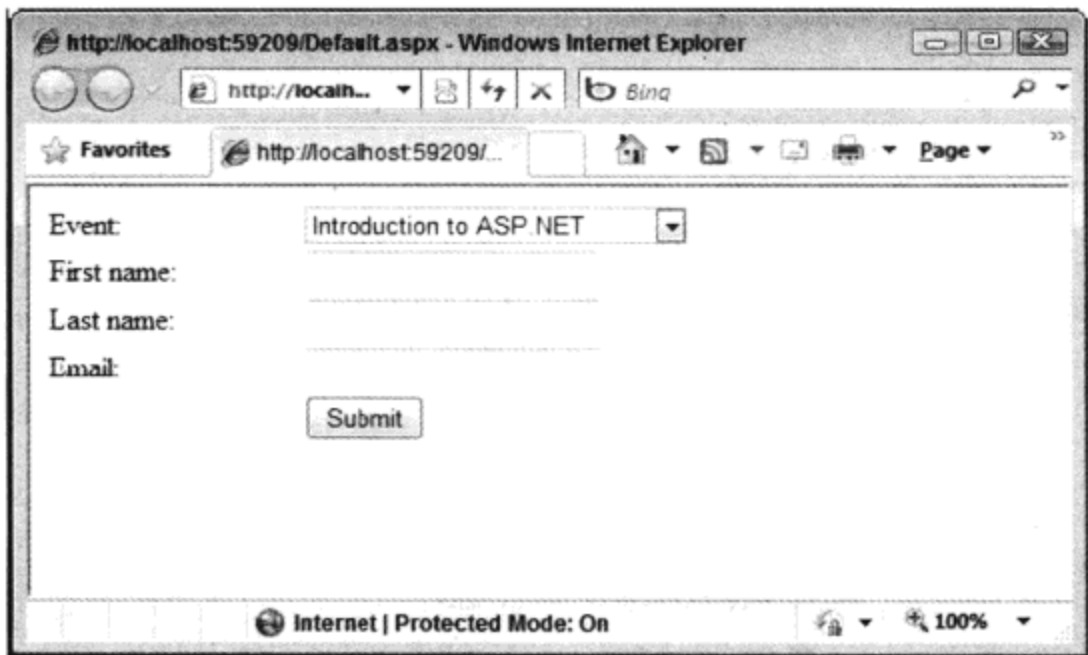


图 18-9

示例的说明

Registration.aspx 文件的第一行是 Page 指令：



可从  
wrox.com  
下载源代码

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Registration.aspx.cs"
    Inherits="EventRegistrationWeb.Registration" %>
```

代码段 Registration.aspx

这个指令定义了要使用的编程语言和类。属性 `AutoEventWireup = "true"` 表示，页面的事件处理程序自动链接到特定方法名上，如稍后所示。`Inherits="EventRegistrationWeb.Registration"` 表示在 ASPX 文件中动态生成的类派生于基类 `Registration`。这个基类位于用 `CodeFile` 属性定义的代码隐藏文件 `Registration.aspx.cs` 中。本章将在后面给 .cs 文件添加处理代码。生成的代码隐藏文件 `Registration.aspx.cs` 如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace EventRegistrationWeb
{
    public partial class Registration : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }
    }
}
```

代码段 Registration.aspx.cs



上面代码中使用的 partial 关键字详见第 10 章。

下面是 ASPX 页面的代码，客户端接收简单的 HTML 代码。页面发送给客户端时，只是从<head>标记中删除了 runat="server"特性。



可从  
wrox.com  
下载源代码

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title></title>
    <style type="text/css">
        .style1
        {
            width: 100%;
        }
    </style>
</head>
<body>
```

代码段 Registration.aspx

还有一些带有 runat="server"特性的 HTML 元素，如<form>元素。通过 runat="server"特性，ASP.NET 服务器控件就会与 HTML 标记关联起来。这个控件可以用于写入服务器端代码。在<form>元素的后面是 System.Web.UI.HtmlControls.HtmlForm 类型的一个对象，该对象有一个用 id 特性定义的变量名 form1。form1 可以用于调用 HtmlForm 类的方法和属性。

HtmlForm 对象创建一个发送给客户端的<form>标记。



可从  
wrox.com  
下载源代码

```
<form id="form1" runat="server">
```

代码段 Default.aspx

当然，不会将 runat 特性发送给客户端。

从工具箱拖放到窗体设计器上的标准控件拥有以<asp:开头的元素<asp:Label>和<asp:DropDownList>。它们是服务器端的 ASP.NET Web 控件，它们与 System.Web.UI.WebControls 名称空间中的 .NET 类相关。<asp:Label>由 Label 类表示，<asp:DropDownList>由 DropDownList 类表示。



可从  
wrox.com  
下载源代码

```
<td>
    <asp:LabelID="labelEvent"runat="server"Text="Event:"></asp:Label>
</td>
<td>
    <asp:DropDownList ID="dropDownListEvents" runat="server">
        <asp:ListItem>SQL Server 2008 and XML</asp:ListItem>
        <asp:ListItem>Office 2007 and XML</asp:ListItem>
        <asp:ListItem>Introduction to ASP.NET</asp:ListItem>
    </asp:DropDownList>
</td>
```

代码段 Registration.aspx

<asp:Label>不向客户端发送<asp:Label>元素，因为它不是一个有效的 HTML 元素，<asp:Label>返回一个<span>标记。同样，<asp:DropDownList>返回<select>元素，<asp:TextBox>返回<input type="text">元素。

ASP.NET 在名称空间 System.Web.UI.HtmlControls 和 System.Web.UI.WebControls 中有 UI 控件类。这两个名称空间有一些类似的控件，也称为 HTML 服务器控件和 Web 服务器控件。例如，HtmlInputText 是 HTML 服务器控件，TextBox 是 Web 服务器控件。HTML 服务器控件提供的方法和属性类似于 HTML 控件，但在 Web 服务器控件中，有更复杂的控件，如 Calendar、DataGrid 和 Wizard。如果不需要在服务器端代码中给控件编程，只需在 JavaScript 中给它编程，就可以只使用 HTML 控件，而不添加 runat="server"特性。

### 18.4 服务器控件

表 18-2 列出了 ASP.NET 提供的一些主要 Web 服务器控件，以及这些控件返回的 HTML 代码。

表 18-2		
控 件	HTML	说 明
Label	<span>	返回一个包含文本的 span 元素
Literal	static text	返回简单的静态文本。使用 Literal 控件，可以根据客户应用程序转换内容
TextBox	<input type="text">	返回 HTML <input type="text">，在该元素中用户可以输入一些值。当文本改变时可以编写服务器端事件处理程序
Button	<input type="submit">	把窗体值发送给服务器


(续表)

控 件	HTML	说 明
LinkButton	<a href="javascript:doPostBack()">	创建一个 anchor 标记, 其中包含给服务器回送的 JavaScript
ImageButton	<input type="image">	生成 image 类型的 input 标记, 显示引用的图像
HyperLink	<a>	创建一个简单的 anchor 标记, 来引用 Web 页面
DropDownList	<select>	创建一个 select 标记, 用户能看到其中的一项, 并可以单击下拉选择框, 从多个项中选择一项
ListBox	<select size="">	创建一个带有 size 特性的 select 标记, 可以一次显示多个项
CheckBox	<input type="checkbox">	返回一个 check box 类型的 input 元素, 显示一个可以选中或取消选中的按钮。如果不使用 CheckBox, 还可以使用 CheckBoxList 创建一个表格, 其中包含多个 check box 元素
RadioButton	<input type="radio">	返回一个 radio 类型的 input 元素。使用单选按钮时, 只能选择组中的一个按钮。与 CheckBoxList 类似, RadioButtonList 也提供了一个按钮列表
Image	<img src="">	返回一个 img 标记, 在客户端上显示 GIF 或 JPG 文件
Calendar	<table>	显示完整的日历, 用户可以在日历中选择日期, 修改月份等。对于输出, 会生成带有 JavaScript 代码的 HTML 表
TreeView	<div><table>	返回一个 div 标记, 其中根据内容包含多个 table 标记。JavaScript 用于打开和关闭客户端上的树

18.5 ASP.NET 回送

Web 服务器控件可以包含服务器上调用的事件处理程序。Button 控件可以包含 Click 事件, DropDownList 控件则提供了 SelectedIndexChanged 事件, TextBox 提供了 TextChanged 事件。

只有进行回送时, 才在服务器上触发事件。文本框中的值改变时, TextChanged 事件不会立即触发, 只有单击 Submit 按钮, 提交了窗体, 并发送给服务器, 才会触发 TextChanged 事件。ASP.NET 运行库会验证控件的状态已改变, 并调用相应的事件处理程序。如果 DropDownList 的选项改变了, 就调用 SelectedIndexChanged 事件, 如果文本框的值改变了, 就调用 TextChanged 事件。



如果希望把更改事件立即传送给服务器(例如, 改变了 DropDownList 的选项), 可以把 AutoPostBack 属性设置为 true。这样就会使用客户端的 JavaScript 把窗体数据立即提交给服务器。当然, 网络通信量也会增加。使用这个功能时要小心。

页面返回给服务器后, 比较控件的新值和旧值是由 View State 完成的。View State 是一个隐藏字段, 它会同页面的内容一起发送给浏览器。当把页面发送给客户端时, View State 会包含与窗体中

控件相同的值。向服务器回送时, View State 也会同控件的新值一起发送给服务器。这样它就可以验证值是否改变, 并调用事件处理程序。


前面的示例应用程序只给客户端发送了一个简单的页面。下面需要处理用户输入的结果。在第一个示例中, 用户输入显示在同一个页面中, 然后使用另一个页面。下面的示例将显示用户输入。

#### 试一试: 显示用户输入

(1) 在 Visual Studio 中, 打开前面创建的 Web 应用程序 EventRegistrationWeb。

(2) 要显示用户输入, 以便进行事件注册, 给 Web 页面 Registration.aspx 添加一个标签 labelResult, 清空这个标签的 Text 属性。

(3) 双击 Submit 按钮, 给它添加 Click 事件处理程序, 在 Registration.aspx.cs 文件中给处理程序添加如下代码:

 可从  
wrox.com  
下载源代码


```
protected void buttonSubmit_Click(object sender, EventArgs e)
{
    string selectedEvent = dropDownListEvents.SelectedValue;
    string firstname = textFirstname.Text;
    string lastname = textLastname.Text;
    string email = textEmail.Text;
    labelResult.Text = String.Format("0) {1} selected the event {2}",
        firstName, lastName, selectedEvent);
}
```

代码段 Registration.aspx.cs

(4) 再次使用 Visual Studio 启动 Web 页面。输入数据, 单击 Submit 按钮后, 该页面会在新标签中显示用户输入。

#### 示例的说明

双击 Submit 按钮, 在 Registration.aspx 文件中给 <asp:Button> 元素添加 OnClick 特性:


 可从  
wrox.com  
下载源代码

```
<asp:Button ID="buttonSubmit" Runat="server" Text="Submit"
    OnClick="buttonSubmit_Click" />
```

代码段 Registration.aspx

对于 Web 服务器控件, OnClick 定义了服务器端的 Click 事件, 在单击按钮时, 就会调用该事件。

在 buttonSubmit\_Click() 方法的实现代码中, 控件的值可以使用属性来读取。dropDownListEvents 是引用 DropDownList 控件的变量。在 ASPX 文件中, ID 设置为 dropDownListEvents, 所以会自动创建变量。SelectedValue 属性返回当前的选择。对于 TextBox 控件, Text 属性返回用户输入的字符串。

 可从  
wrox.com  
下载源代码

```
string selectedEvent = dropDownListEvents.SelectedValue;
string firstName = textFirstName.Text;
string lastName = textLastName.Text;
string email = textEmail.Text;
```

代码段 Registration.aspx.cs

labelResult 标签的 Text 属性设置了结果:



可从  
wrox.com  
下载源代码

```
labelResult.Text = String.Format("{0} {1} selected the event {2}",
    firstName, lastName, selectedEvent);
```

代码段 Registration.aspx.cs

ASP.NET 不是在同一页面上显示结果，而是在另一个页面上显示，如下面的示例所示。

### 试一试：在第二个页面上显示结果

- (1) 创建一个新的 WebForm，命名为 ResultsPage.aspx。
- (2) 给 ResultsPage 添加一个标签 labelResult。
- (3) 给 ResultsPage 类的 Page\_Load 方法添加代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Web.UI.WebControls;

namespace EventRegistrationWeb
{
    public partial class ResultsPage : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            try
            {
                DropDownList dropDownListEvents =
                    (DropDownList)PreviousPage.FindControl("dropDownListEvents");
                string selectedEvent = dropDownListEvents.SelectedValue;
                string firstName = ((TextBox)PreviousPage.FindControl(
                    "textFirstName")).Text;
                string lastName = ((TextBox)PreviousPage.FindControl(
                    "textLastName")).Text;
                string email = ((TextBox)PreviousPage.FindControl(
                    "textEmail")).Text;
                labelResult.Text = String.Format("{0} {1} selected the event {2}",
                    firstName, lastName, selectedEvent);
            }
            catch
            {
                labelResult.Text = "The originating page must contain " +
                    "textFirstName, textLastName, textEmail controls";
            }
        }
    }
}
```

代码段 ResultsPage.aspx.cs

- (4) 把 Registration.aspx 页面上 Submit 按钮的 PostbackUrl 属性设置为 ResultsPage.aspx。
- (5) 可以删除 Submit 按钮的 Click 事件处理程序，因为不再需要它了。
- (6) 启动 Default.aspx 页面，填充一些数据，单击 Submit 按钮，就会重定向到 ResultsPage.aspx 页面，其中显示了输入的数据。



示例的说明

在 ASP.NET 中, Button 控件有一个新属性 PostbackUrl, 定义了应从 Web 服务器上请求的页面。这个属性创建客户端 JavaScript 代码, 用 Submit 按钮的客户端 onclick 处理程序请求所定义的页面。

```
<input type="submit" name="buttonSubmit" value="Submit"
  onclick="javascript:webForm_DoPostBackWithOptions(
    new WebForm_PostBackOptions("&quot;buttonSubmit&quot;;,
      &quot;&quot;;, false, &quot;&quot;;, &quot;ResultPage.aspx&quot;;,
        false, false))" id="buttonSubmit" />
```

浏览器把第一个页面中窗体的所有数据都发送到新页面上, 但是, 在新请求的页面上, 需要从前面页面定义的控件中获取数据。为了访问前面页面中的控件, Page 类定义了属性 PreviousPage。PreviousPage 返回一个 Page 对象, 这个页面的控件可以使用 FindControl()方法来访问。FindControl()定义为返回一个 Control 对象, 所以必须把返回值的类型转换为所搜索的控件类型。



可从  
WTOX.COM  
下载源代码

```
DropDownList dropDownListEvents =
  (DropDownList)PreviousPage.FindControl("dropDownListEvents");
```

代码段 ResultsPage.aspx.cs

在开发过程中, 如果不使用 FindControl()方法访问前面页面的值, 就可以把对前面页面的访问强类型化, 以减少错误。为此, 应使用 default\_aspx 类的一个属性返回一个定制结构, 如下面的示例所示。

试一试: 创建强类型化的 PreviousPage

- (1) 选择 Project | Add New Class, 为项目添加一个新类, 命名为 RegistrationInfo。
- (2) 实现 RegistrationInfo 类, 如下所示:



可从  
WTOX.COM  
下载源代码

```
public class RegistrationInfo
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Email { get; set; }
  public string SelectedEvent { get; set; }
}
```

代码段 RegistrationInfo.cs

- (3) 在 Registration.aspx.cs 文件中给 Registration 类添加公共属性 RegistrationInfo:



可从  
WTOX.COM  
下载源代码

```
public RegistrationInfo RegistrationInfo
{
  get
  {
    return new RegistrationInfo
    {
      FirstName = textFirstName.Text,
      LastName = textLastName.Text,
      Email = textEmail.Text,
    }
  }
}
```

```

        SelectedEvent = dropDownListEvents.SelectedValue
    };
}
}

```

代码段 Registration.aspx.cs

(4) 在 ResultPage.aspx 文件的 Page 指令下面添加 PreviousPageType 指令:



可从  
wrox.com  
下载源代码

```

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="ResultPage.aspx.cs"
    Inherits="ResultPage_aspx" %>
<%@ PreviousPageType VirtualPath="~/Registration.aspx" %>

```

代码段 ResultPage.aspx

(5) 在 ResultsPage 类的 Page\_Load() 方法中, 代码可以简化为:



可从  
wrox.com  
下载源代码

```

protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        RegistrationInformation ri = PreviousPage.RegistrationInformation;

        labelResult.Text = String.Format("{0} {1} selected the event {2}",
            ri.FirstName, ri.LastName, ri.SelectedEvent);
    }
    catch
    {
        labelResult.Text = "The originating page must contain " +
            "textFirstName, textLastName, textEmail controls";
    }
}

```

代码段 ResultPage.aspx.cs

### 示例的说明

PreviousPageType 指令创建了一个 PreviousPage 类型的属性, 它返回与该指令关联的类型。在其实现代码中, 调用了基类的 PreviousPage 属性, 如下面的代码所示:

```

public new EventRegistrationWeb.Default PreviousPage {
    get {
        return ((EventRegistrationWeb.Default) (base.PreviousPage));
    }
}

```

这里没有使用 PreviousPageType 指令的 VirtualPath 特性定义上一个页面的类型, 而使用了 TypeName 特性。如果前面有多个页面, 就可以使用这个特性。此时, 需要为前面的所有页面定义一个基类, 并把该基类赋予 TypeName 特性。

## 18.6 ASP.NET AJAX 回送

在一般的 ASP.NET 回送中，会请求整个页面。回送用户已经加载的同一个页面时，也会再次返回整个页面。为了减少网络上的传输量，可以使用 ASP.NET Ajax 回送。在 Ajax 回送中，只使用 JavaScript 返回并刷新页面的一部分，使用 UpdatePanel 可以方便地做到这一点。

为了便于比较 ASP.NET 回送和 ASP.NET Ajax 回送，下面的示例将当前时间输出到两个标签中，其中一个标签在 UpdatePanel 中，另一个不在 UpdatePanel 中。

### 试一试：使用 UpdatePanel 控件

- (1) 使用 Visual Studio 打开前面创建的项目 EventRegistrationWeb。
- (2) 给现有的网站添加一个新 AJAX Web 窗体 UpdatePanelDemo.aspx。
- (3) 在工具箱的 AJAX Extensions 类别中，给页面添加一个 UpdatePanel 控件。
- (4) 在 UpdatePanel 控件内部添加一个标签和一个按钮，在 UpdatePanel 控件外部再添加一个标签和一个按钮，把 UpdatePanel 控件内部和外部的按钮的 Text 属性分别设置为 AJAX Postback 和 ASP.NET Postback。



```
<form id="form1" runat="server">
<div>
    <asp:ScriptManager ID="ScriptManager1" runat="server">
    </asp:ScriptManager>
</div>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <asp:Label ID="Label1" runat="server" Text="Label"> </asp:Label>
        <asp:Button ID="Button1" runat="server" Text="AJAX Postback" />
        <asp:Button ID="Button1" runat="server" Text="AJAX Postback" />
        <asp:Button Click="OnButtonClick" />
    </ContentTemplate>
</asp:UpdatePanel>
<asp:Label ID="Label2" runat="server" Text="Label"> </asp:Label>
<asp:Button ID="Button2" runat="server" Text="ASP.NET Postback" />
</form>
```

代码段 UpdatePanelDemo.aspx

- (5) 给两个按钮指定 Click 事件处理程序 OnButtonClick(), 其实现代码如下所示:



```
protected void OnButtonClick(object sender, EventArgs e)
{
    DateTime now = DateTime.Now;
    Label1.Text = now.ToString();
    Label2.Text = now.ToString();
}
```

代码段 UpdatePanelDemo.aspx.cs

- (6) 启动应用程序，单击两个按钮。单击 AJAX Postback 按钮仅刷新第一个标签。单击 ASP.NET Postback 会刷新整个页面，如图 18-10 所示。

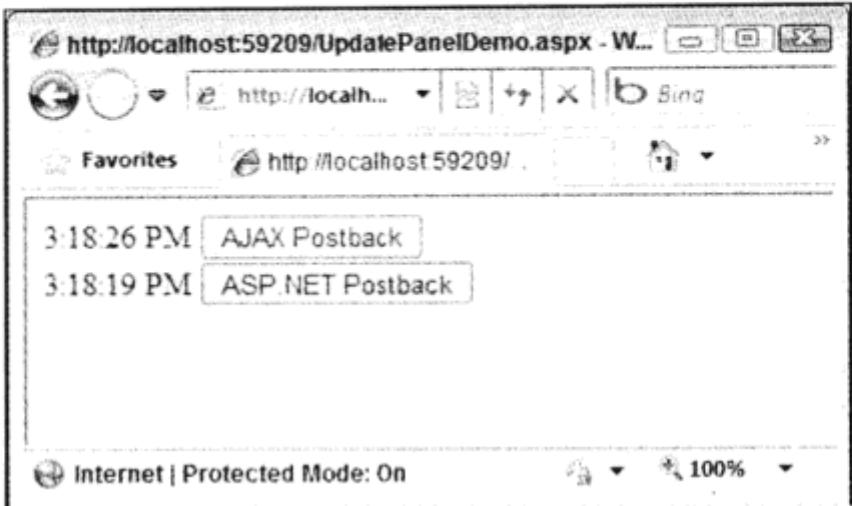


图 18-10

示例的说明

在 ASP.NET AJAX 页面上，需要一个 ScriptManager 对象。这个对象使用 AJAX Web Form 模板添加。ScriptManager 类加载了包含几个功能的 JavaScript 函数。也可以使用这个类加载自己的定制脚本。ScriptManager 的属性如表 18-3 所示。

表 18-3

属 性	说 明
EnablePageMethods	指定在 ASPX 页面上定义的公共静态方法是否可以从客户脚本中作为 Web 服务方法调用
EnablePartialRendering	为了启用 UpdatePanel 的部分呈现功能，这个属性必须设置为 true(默认)
LoadScriptsBeforeUI	指定脚本放在所返回的 HTML 页面的什么地方。把它们放在<head>元素内部，脚本就会在加载 UI 之前加载
ScriptMode	指定应使用脚本的调试版本还是发布版本
ScriptPath	指定定制脚本所在的目录的根路径
Scripts	包含应呈现在客户端的定制脚本文件集合
Services	包含可以从客户脚本中调用的 Web 服务引用集合

页面上的 ASP.NET 控件 Button 让客户机创建 HTML Submint 按钮。Button2 给服务器发送一个一般的 HTTP POST 请求。Button1 在 UpdatePanel 的内部，所以客户脚本关联上该按钮的 Click 事件，发出一个 Ajax POST 请求。Ajax POST 请求使用 XmlHttpRequest 对象给服务器发送一个请求。服务器只返回更新 UI 所需的数据。解释了数据后，JavaScript 代码就修改 UpdatePanel 内部的 HTML 控件，显示一个新的 UI。

一个页面上可以有多个 UpdatePanel。只需在一个页面上添加多个 UpdatePanel，每个 UpdatePanel 就会在 Ajax POST 请求时更新。更新可以用触发器控制，如下面的示例所示。

试一试：使用触发器更新面板

- (1) 在 Visual Studio 中打开前面创建的 EventRegistrationWeb 项目。
- (2) 在现有网站上添加一个新的 AJAX Web 窗体 UpdatePanelWithTrigger.aspx。

- (3) 添加两个 UpdatePanel 控件。
- (4) 在每个 UpdatePanel 控件中添加一个 Label 和一个 Button 控件。
- (5) 把两个 Button 控件的 Click 事件处理程序赋予 OnButtonClick()方法, 实现该方法的代码如下:



```
protected void OnButtonClick(object sender, EventArgs e)
{
    DateTime now = DateTime.Now;
    Label1.Text = now.ToLongTimeString();
    Label2.Text = now.ToLongTimeString();
}
```

代码段 UpdatePanelWithTrigger.aspx.cs

- (6) 运行应用程序。无论单击哪个按钮, 两个标签都会改变, 如图 18-11 所示。



图 18-11

- (7) 把两个 UpdatePanel 控件的 UpdateMode 属性都从 Always 改为 Conditional。
- (8) 再次运行应用程序。现在单击一个按钮, 就只有该按钮所在的 UpdatePanel 控件中的标签会改变。
- (9) 选择第一个 UpdatePanel, 单击 Triggers 属性旁边的省略号, 打开如图 18-12 所示的对话框。添加一个 AsynchronousPostBack 触发器, 把 ControlID 属性设置为第二个 UpdatePanel 中的按钮 Button2, 把 EventName 设置为 Click。

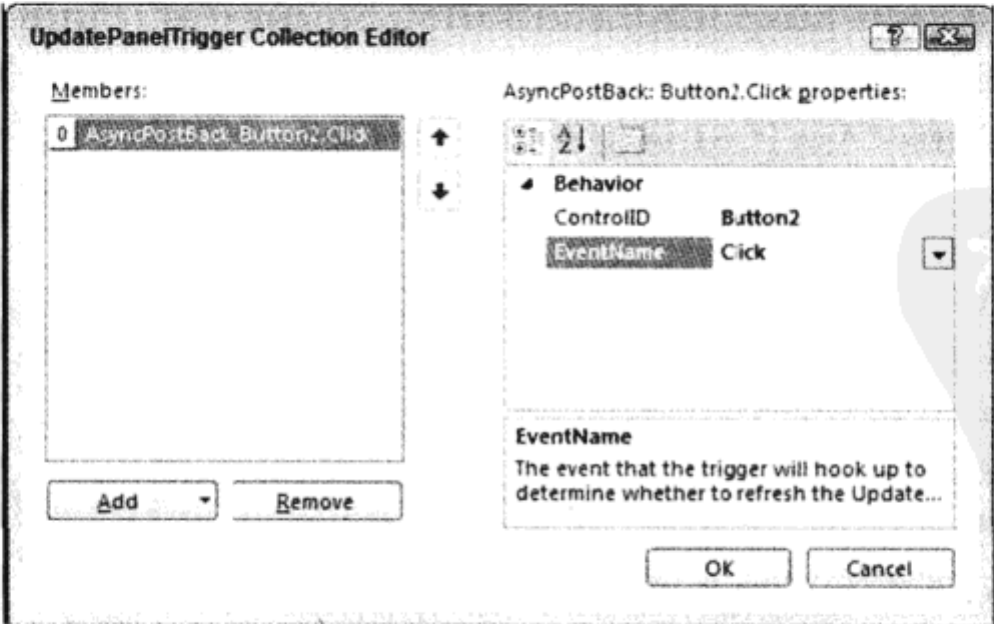


图 18-12

(10) 运行应用程序。单击 Button2，会更新两个 UpdatePanel 的内容，而单击 Button1，只更新第一个 UpdatePanel 控件的内容。

示例的说明

可以改变 UpdatePanel 的更新操作。它默认为每次发生 Ajax 回送事件时更新。可以改变这个更新操作，只有更新事件发生在 UpdatePanel 内部时才更新控件，或者定义一个触发器，由 UpdatePanel 外部的控件引发更新操作。

下面的 ASPX 代码定义了 UpdatePanel1 的 AsyncPostBackTrigger:



```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" UpdateMode="Conditional">
  <ContentTemplate>
    <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
    <asp:Button ID="Button1" runat="server" Text="Button"
      OnClick="OnButtonClick" />
  </ContentTemplate>
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Button2" EventName="Click" />
  </Triggers>
</asp:UpdatePanel>
```

代码段 UpdatePanelWithTrigger.aspx

表 18-4 中描述了 UpdatePanel 控件的属性。

表 18-4

属 性	说 明
ChildrenAsTriggers	如果这个属性设置为 true，则 UpdatePanel 的子控件执行回送时，会更新 UpdatePanel 的内容
RenderMode	指定 UpdatePanel 的呈现方式。其值可以是 UpdatePanelRenderMode.Block 或 UpdatePanelRenderMode.Inline。Block 枚举值指定应呈现<div>标记，而 Inline 指定显示<span>标记
UpdateMode	设置为 UpdatePanelUpdateMode 的一个枚举值。Always 会在每个 Ajax 回送时更新 UpdatePanel，Conditional 则取决于触发器
Triggers	指定 AsyncPostBackTrigger 和 PostbackTrigger 元素的集合，指定了更新 UpdatePanel 内容的时间

18.7 输入的有效性验证

在用户输入数据时，应检查数据是否有效。这个检查可以在客户端和服务端上进行。在客户端上检查数据可以使用 JavaScript 来进行。但是，如果使用 JavaScript 在客户端检查数据，就一定要在服务端上也进行检查，因为客户端是永远不能完全信任的。可以在浏览器上禁用 JavaScript，但黑客能使用其他 JavaScript 函数，接收不正确的输入。在服务器上检查数据是绝对必须的。在客户端上检查数据还能提供更好的性能，因为在客户端上验证完数据之前，不需要把数据发送到服务器上。

在 ASP.NET 中，不需要自己编写验证函数。许多已有的验证控件能进行客户端和服务端验证。

下面这个示例显示了与文本框 `textFirstname` 相关的验证控件 `RequiredFieldValidator`。所有的验证控件都有 `ErrorMessage` 和 `ControlToValidate` 属性。如果输入不正确，`ErrorMessage` 可以定义显示的消息。默认的错误消息显示在验证控件所在的位置上。`ControlToValidate` 属性定义对输入进行检查的控件。

```
<asp:TextBox ID="textFirstname" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
    ErrorMessage="Enter your first name" ControlToValidate="textFirstname">
</asp:RequiredFieldValidator>
```

表 18-5 列出了所有的验证控件。

表 18-5	
控 件	说 明
RequiredFieldValidator	指定所验证的控件需要输入一些内容。如果所验证的控件要设置初始值，而用户必须改变初始值，就可以使用验证控件的 <code>InitialValue</code> 属性设置这个初始值
RangeValidator	定义允许用户输入的最大值和最小值。该控件的特定属性有 <code>MinimumValue</code> 和 <code>MaximumValue</code>
RegularExpressionValidator	通过 <code>ValidationExpression</code> 属性，可以使用 Perl 5 语法设置一个正则表达式，来检查用户的输入
CompareValidator	比较多个值(如密码)。这个验证控件不仅可以比较两个值是否相等，还可以使用 <code>Operator</code> 属性设置多个选项。 <code>Operator</code> 属性是 <code>ValidationCompareOperator</code> 类型，定义了枚举值，如 <code>Equal</code> 、 <code>NotEqual</code> 、 <code>GreaterThan</code> 和 <code>DataTypeCheck</code> 。使用 <code>DataTypeCheck</code> 可以检查输入值，确定它是否是指定的数据类型，例如，查看输入的日期是否正确
CustomValidator	如果其他验证控件都不满足验证要求，就可以使用 <code>CustomValidator</code> 。通过它可以定义客户端和服务端验证功能
ValidationSummary	编写页面的小结，而不是编写与输入控件直接相关的错误消息

在前面的示例应用程序中，用户可以输入姓、名和电子邮件地址。下面将使用验证控件扩展这个应用程序。

试一试：检查需要的输入和电子邮件地址

- (1) 打开前面在 Visual Studio 中创建的项目 `EventRegistrationWeb`。
- (2) 打开文件 `Registration.aspx`。
- (3) 在编辑器的设计视图中选择右边的列，再选择菜单项 `Table | Insert | Column to the Right`，给表添加一个新列。
- (4) 需要输入姓、名和电子邮件地址。应检查电子邮件地址的语法是否正确。添加 3 个 `RequiredFieldValidator` 控件和 1 个 `RegularExpressionValidator` 控件，如图 18-13 所示。



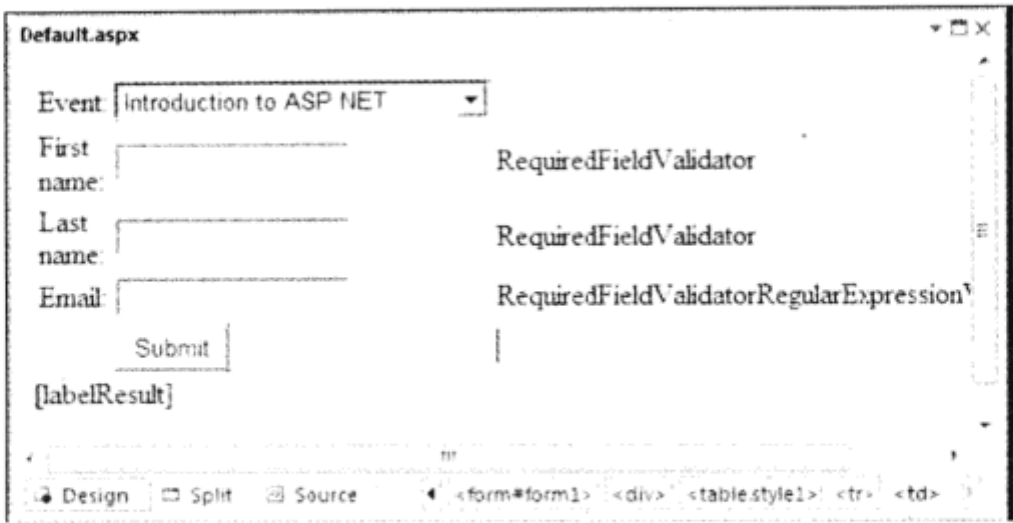


图 18-13

(5) 按照表 18-6 中的定义配置验证控件。

表 18-6

验证控件	属 性	值
RequiredFieldValidator	ErrorMessage	Firstname is required.
	ControlToValidate	textFirstName
RequiredFieldValidator	ErrorMessage	Last name is required.
	ControlToValidate	textLastName
RequiredFieldValidator	ErrorMessage	Email is required.
	ControlToValidate	textEmail
	Display	Dynamic
RegularExpressionValidator1	ErrorMessage	Enter a valid email.
	ControlToValidate	textEmail
	ValidationExpression	\w+([-+.'\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*
	Display	Dynamic

(6) 不需要手工输入正则表达式，而可以在 Properties 窗口中单击 ValidationEpression 的省略号按钮，启动 Regular Expression Editor，如图 18-14 所示。这个编辑器提供了一些预定义的正则表达式，包括用来选择检查 Internet e-mail address 的正则表达式。

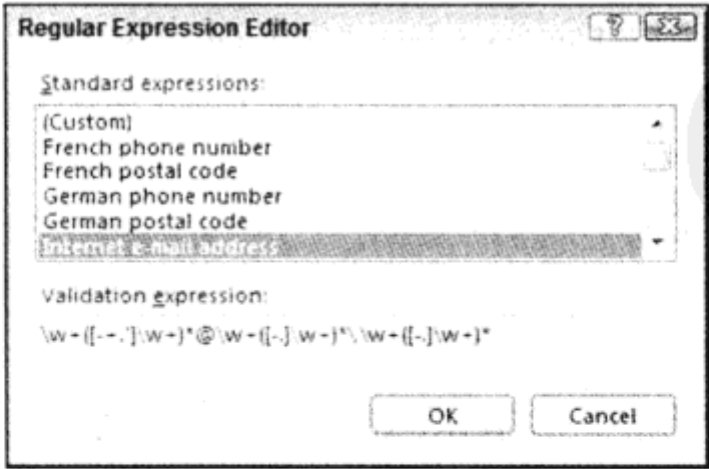



图 18-14

(7) 如果回送的页面不同于包含验证控件的页面(使用前面设置的 `PostBackUrl` 属性), 则在新的页面中必须使用 `IsValid` 属性, 验证上一个页面的结果是否有效。把以下代码添加到 `ResultsPage` 类的 `Page_Load` 方法中:



可从  
wrox.com  
下载源代码

```
protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        if (!PreviousPage.IsValid)
        {
            labelResult.Text = "Error in previous page";
            return;
        }
        //...
    }
}
```

代码段 `ResultsPage.aspx.cs`

(8) 现在就可以启动应用程序了。如果没有输入数据或数据不正确, 验证控件就会显示错误消息, 如图 18-15 所示。

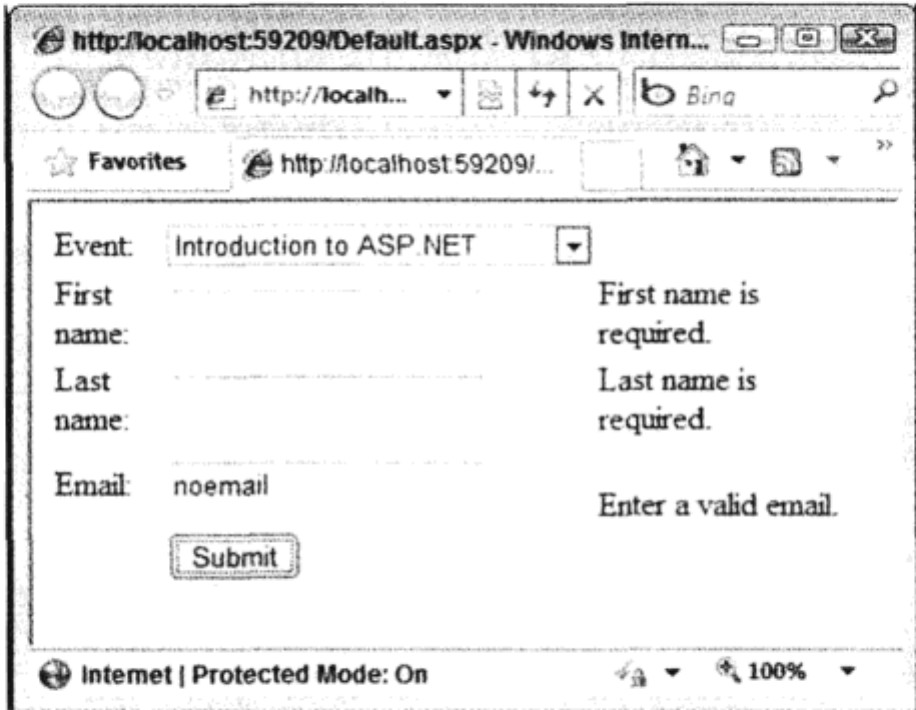


图 18-15

示例的说明

验证控件创建了客户端 JavaScript 代码, 在客户端上验证输入, 还创建了服务器端代码, 在服务器上验证输入。把验证属性 `EnableClientScript` 设置为 `false`, 就可以关闭 JavaScript。设置 `Page` 类的 `ClientTarget` 属性, 也可以关闭 JavaScript, 而不需要对每个验证控件执行改变属性值的操作。

根据客户的类型, ASP.NET 控件可以向客户端返回 JavaScript, 也可以不返回 JavaScript。这取决于 `ClientTarget` 属性, 该属性默认设置为 `automatic`, 此时, 根据 Web 浏览器的功能, 返回或不返回脚本代码。如果 `ClientTarget` 设置为 `downlevel`, 脚本代码就不返回给任何客户端, 而把属性 `ClientTarget` 设置为 `uplevel`, 就总是返回脚本代码。

`ClientTarget` 属性的设置可以在 `Page` 类的 `Page_Load()` 方法中进行:

```
protected void Page_Load(object sender, EventArgs e)
```

```
{
    ClientTarget = "downlevel";
}
```

18.8 状态管理

HTTP 协议是无状态的。从客户端到服务器的连接可以在每个请求之后关闭。但是一般需要把一些客户端信息从一个页面传送给另一个页面。这可以通过几种方式来实现。

在保存状态的各种方式中，主要区别是：状态是存储在客户端还是服务器上。表 18-7 中简要列出了各种状态管理技术以及状态保持有效的时间。

表 18-7

状 态 类 型	客户端或服务器资源	有 效 时 间
ViewState	客户端	只在一个页面中
Cookie	客户端	关闭浏览器时会删除临时 cookie，永久 cookie 存储在客户系统的磁盘上
Session	服务器	会话状态与浏览器会话相关。会话在超时（默认为 20 分钟）后变得无效
Application	服务器	在所有的客户端上共享应用程序状态，这个状态在服务器重新启动之前都是有效的
Cache	服务器	类似于应用程序状态，缓存是共享的。但是，使缓存无效有更好的控制方式

下面详细介绍这些技术。

18.8.1 客户端的状态管理

首先介绍客户端的状态管理：ViewState 和 cookie。

1. ViewState

前面已讨论过在客户端存储状态的一种技术：ViewState。Web 服务器控件自动使用 ViewState 来使事件工作。ViewState 包含的状态与控件发送给客户端时包含的状态相同。当浏览器把窗体发送回服务器时，ViewState 包含了初始值，但所发送的控件包含新值。如果初始值和新值有区别，就调用相应的事件处理程序。

使用 ViewState 的缺点是，数据总是要从服务器传送给客户端，再从客户端传送给服务器，增加了网络流量。为了减少网络流量，可以关闭 ViewState。在 Page 指令中把 EnableViewState 属性设置为 false，就可以关闭页面中所有控件的 ViewState。

```
<%@ Page Language="C#" AutoEventWireUp="true" CodeFile="Registration.aspx.cs"
    Inherits="Registration_aspx" EnableViewState="false" %>
```

设置一个控件的 EnableViewState 属性，也可以配置该控件的 ViewState。无论页面进行了什么

配置, 只要定义了控件的 `EnableViewState` 属性, 就使用相应的控件值。只有没有配置 `ViewState` 的控件才使用页面配置的值。

还可以把定制的数据存储在 `ViewState` 中。为此可以使用索引符和 `Page` 类的 `ViewState` 属性。可以使用 `index` 参数定义一个名称, 用于访问 `ViewState` 值。

```
ViewState["mydata"] = "my data";
```

可以读取前面存储的 `ViewState`, 如下所示:

```
string mydata = (string)ViewState["mydata"];
```

在发送给客户端的 HTML 代码中, 整个页面的 `ViewState` 存储在一个隐藏字段中:

```
<input type="hidden" name="__VIEWSTATE"
value="/wEPDwUKLTU4NzY5NTcwNw8WAh4HbXlzdGF0ZQUFbXl2YWwWAglDD2QWAg
IFDw8WAh4EYGV4dAUFBXl2YWxkZGTCdCwUOcAW97aKpcjtl1tzJ7ByUA==" />
```

使用隐藏字段的优点是, 每个浏览器都可以使用这个特性, 用户不能关闭它。

`ViewState` 只保存在页面中。如果状态应保存在多个不同的页面中, 就应使用 `cookie` 在客户端保存状态。

## 2. cookie

`cookie` 在 HTTP 头中定义。使用 `HttpResponse` 类可以把 `cookie` 发送给客户端。`Response` 是 `Page` 类的一个属性, 它返回一个 `HttpResponse` 类型的对象。`HttpResponse` 类定义了返回 `HttpCookieCollection` 的 `Cookies` 属性。使用 `HttpCookieCollection` 可以向客户端返回多个 `cookie`。

下面的示例代码说明了如何把 `cookie` 发送给客户端。首先实例化一个 `HttpCookie` 对象。在这个类的构造函数中, 设置 `cookie` 的名称, 这里是 `mycookie`。`HttpCookie` 类的 `Values` 属性可以添加多个 `cookie` 值。如果只需要返回一个 `cookie` 值, 就可以使用 `Value` 属性。但如果要发送多个 `cookie` 值, 最好把值添加到一个 `cookie` 中, 而不是使用多个 `cookie`。

```
string myval = "myval";
HttpCookie cookie = new HttpCookie("mycookie");
cookie.Values.Add("mystate", myval);
Response.Cookies.Add(cookie);
```

`cookie` 可以是临时的, 仅在一个浏览器会话中有效, 也可以存储在客户端的磁盘上。为了使 `cookie` 变成永久的, 必须使用 `HttpCookie` 对象设置 `Expires` 属性。使用 `Expires` 属性可以定义 `cookie` 不再有效的日期, 这里把它设置为三个月后。

```
Var cookie = new HttpCookie("mycookie");
cookie.Values.Add("mystate", "myval");
cookie.Expires = DateTime.Now.AddMonths(3);
Response.Cookies.Add(cookie);
```

尽管可以设置特定的日期, 但无法保证 `cookie` 会一直存储到该日期为止。用户可以删除 `cookie`, 如果在本地存储了太多的 `cookie`, 浏览器应用程序也会删除 `cookie`。浏览器只能给一个服务器存储 20 个 `cookie`, 给所有的服务器存储 300 个 `cookie`。达到这个界限时, 就会删除一段时间内不再使用的 `cookie`。

客户从服务器上请求一个页面时, 这个服务器的 cookie 就可以在客户端上使用, 并作为 HTTP 请求的一部分发送给服务器。要在 ASP.NET 页面中读取 cookie, 可以访问 `HttpRequest` 对象中的 cookie 集合。

与 HTTP 响应一样, `Page` 类也有一个 `Request` 属性返回 `HttpRequest` 类型的对象。`Cookies` 属性返回 `HttpCookieCollection`, 它可以用于读取客户端发送的 cookie。可以用索引符通过其名称来访问 cookie, 然后使用 `HttpCookie` 的 `Values` 属性从 cookie 中获取值。

```
HttpCookie cookie = Request.Cookies["mycookie"];
string myval = cookie.Values["mystate"];
```

ASP.NET 允许更加方便地使用 cookie。但必须了解 cookie 的限制。如前所述, 浏览器只能接收来自一个服务器的 20 个 cookie, 以及来自所有服务器的 300 个 cookie。另外, cookie 存储的数据量不能超过 4K。这些限制使客户端磁盘不会被 cookie 占满。

## 18.8.2 服务器端的状态管理

除了在客户端上保存状态之外, 还可以在服务器上保存状态。使用客户端的状态, 其缺点在于增加了数据在网络之间的传送。使用服务器端状态的缺点在于, 服务器必须给其客户端分配资源。下面详细讨论服务器端的状态管理技术。

### 1. 会话

会话状态与浏览器会话相关。客户在服务器上第一次打开 ASP.NET 页面时, 会话就开始了。当客户在 20 分钟之内没有访问服务器时, 会话结束。

可以在 `Global Application` 类中定义自己的代码, 在会话开始或结束时运行。`Global Application` 类使用菜单项 `Project | Add New Item | Global Application Class` 创建。在新的 `Global Application` 类中, 会创建 `global.asax` 文件。在这个文件中, 在派生于基类 `HttpApplication` 的 `Global` 类中定义了一些事件处理程序例程:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;

namespace EventRegistrationWeb
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
        }

        protected void Session_Start(object sender, EventArgs e)
        {
            // Code that runs when a new session is started
        }

        protected void Application_BeginRequest(object sender, EventArgs e)
        {

```

```

    }

    protected void Application_AuthenticateRequest(object sender, EventArgs e)
    {

    }

    protected void Application_Error(object sender, EventArgs e)
    {
        // Code that runs when an unhandled error occurs
    }

    protected void Session_End(object sender, EventArgs e)
    {
        // Code that runs when a session ends.
        // Note: The Session_End event is raised only when the session state
        // mode is set to InProc in the Web.config file. If session mode is
        // set to StateServer or SQLServer, the event is not raised.
    }

    protected void Application_End(object sender, EventArgs e)
    {
        // Code that runs on application shutdown
    }
}

```

---

代码段 Global.asax.cs

---

会话状态可以存储在 `HttpSessionState` 对象中。可以使用 `Page` 类的 `Session` 属性来访问与当前 HTTP 上下文相关的会话状态对象。在 `Session_Start()` 事件处理程序中，可以初始化会话变量。在下面的示例中，名为 `mydata` 的会话状态被初始化为 0:

```

void Session_Start(Object sender, EventArgs e) {
    // Code that runs on application startup
    Session["mydata"] = 0;
}

```

可以使用会话状态名读取会话状态，这通过 `Session` 属性来完成:

```

void Button1_Click(object sender, EventArgs e)
{
    int val = (int)Session["mydata"];
    Label1.Text = val.ToString();
    val += 4;
    Session["mydata"] = val;
}

```

要把客户端与其会话变量关联起来，ASP.NET 默认使用一个临时 cookie 和一个会话标识符。ASP.NET 也支持没有 cookie 的会话，其中的 URL 标识符用于把 HTTP 请求映射到同一个会话。

## 2. 应用程序

如果应在多个客户端之间共享数据, 就可以使用应用程序状态。应用程序状态的使用方式与会话状态非常类似。对于应用程序状态, 应使用 `HttpApplicationState` 类, 通过 `Page` 类的 `Application` 属性可以访问它。

在下面的示例中, 在启动 Web 应用程序时, 初始化应用程序变量 `userCount`。`Application_Start()` 是 `global.asax` 文件中的事件处理方法, 在启动网站的第一个 ASP.NET 页面时调用该方法。这个变量用于统计访问网站的用户数。

```
void Application_Start(Object sender, EventArgs e) {
    // Code that runs on application startup
    Application["userCount"] = 0;
}
```

在 `Session_Start()` 事件处理程序中, 应用程序变量 `userCount` 的值会递增。在改变应用程序变量之前, 必须用 `Lock()` 方法锁定应用程序对象, 否则就会出现线程问题, 因为多个客户可以同时访问一个应用程序变量。在改变了应用程序变量的值后, 还必须调用 `Unlock()` 方法。注意锁定和解锁之间的时间非常短暂, 在此时间段内, 不应读取文件或数据库中的数据。否则, 其他客户就必须等到数据访问完成之后才能操作。

```
void Session_Start(Object sender, EventArgs e) {
    // Code that runs when a new session is started
    Application.Lock();
    Application["userCount"] = (int)Application["userCount"] + 1;
    Application.Unlock();
}
```

读取应用程序状态中的数据与读取会话状态中的数据一样简单:

```
Label1.Text = this.Application["userCount"].ToString();
```

不要在应用程序状态中存储太多的数据, 因为应用程序状态需要占用服务器资源, 直到服务器停止或重新启动之后, 才会释放这些资源。

## 3. 缓存

缓存是服务器端状态, 它类似于应用程序状态, 因为它在所有的客户端上共享。缓存与应用程序状态的区别是, 缓存要灵活得多: 可以通过多种方式来定义状态的失效时间。我们不是给每个请求读取文件或数据库, 而是把数据存储在缓存中。

对于缓存, 需要使用 `System.Web.Caching` 名称空间和 `Cache` 类。给缓存添加对象的过程如下:

```
Cache.Add("mycache", myobj, null, DateTime.MaxValue,
    TimeSpan.FromMinutes(10), CacheItemPriority.Normal, null);
```

`Page` 类的 `Cache` 属性返回一个 `Cache` 对象。使用 `Cache` 类的 `Add()` 方法, 可以把任意对象赋予缓存。`Add()` 方法的第一个参数定义了缓存项的名称。第二个参数是应缓存的对象。第三个参数定义了依赖关系, 例如, 缓存项可以依赖于一个文件: 当文件改变时, 缓存对象就会失效。在上面的示



例中，没有定义依赖关系，因为这个参数设置为 `null`。

第四和五个参数用于设置缓存项的有效时间。第四个参数定义了缓存项失效的绝对时间。第五个参数定义了使缓存项失效的相对时间(即从最后一次访问以来经历的时间)。上面的示例使用了相对时间，如果缓存项 10 分钟以来未受到访问，就会失效。

第六个参数定义了缓存的优先级。`CacheItemPriority` 是一个设置缓存优先级的枚举。如果 ASP.NET 辅助进程有很高的内存利用率，ASP.NET 运行库就根据优先级删除缓存项。优先级较低的项先删除。最后一个参数用于定义一个方法，在删除缓存项时调用该方法。当缓存依赖于一个文件时，就可以使用最后一个参数：当文件改变时，就删除缓存项，调用事件处理程序。通过这个事件处理程序，可以再次读取文件，重新加载缓存。

使用前面介绍的索引符和会话状态或应用程序状态，就可以读取缓存项。在使用从 `Cache` 属性返回的对象之前，必须检查结果是否为 `null`，当缓存失效时，结果就是 `null`。如果从 `Cache` 索引符返回的值不是 `null`，就可以对返回的对象进行类型转换，用于存储缓存项：

```
object o = Cache["mycache"];
if (o == null)
{
    // Reload the cache.
}
else
{
    // Use the cache.
    MyClass myObj = (MyClass)o;
    //...
}
```

## 18.9 样式

Visual Studio 支持使用层叠样式表(Cascading Style Sheets, CSS)给 Web 页面指定样式。使用 CSS 可以定义 HTML 页面的外观和格式。使用 CSS 无需定制每个 HTML 元素，而可以为特定的元素定义样式(参见下面的示例)，再按名称引用它们，以便于重用。

试一试：给元素定义样式

- (1) 打开前面创建的 Web 应用程序项目 `EventRegistrationWeb`。
- (2) 选择 `Project | New Folder`，添加一个新文件夹 `Styles`。
- (3) 在 `Solution Explorer` 中选择这个文件夹，再选择 `Project | Add New Item`，之后选择 `Style Sheet`，创建一个新的样式表，命名为 `Site.css`。
- (4) 这个样式表默认包含一个空的 `body` 元素。
- (5) 在 `body` 元素的花括号内部单击，打开关联菜单，选择 `Build Style`，打开如图 18-16 所示的 `Modify Style` 对话框。

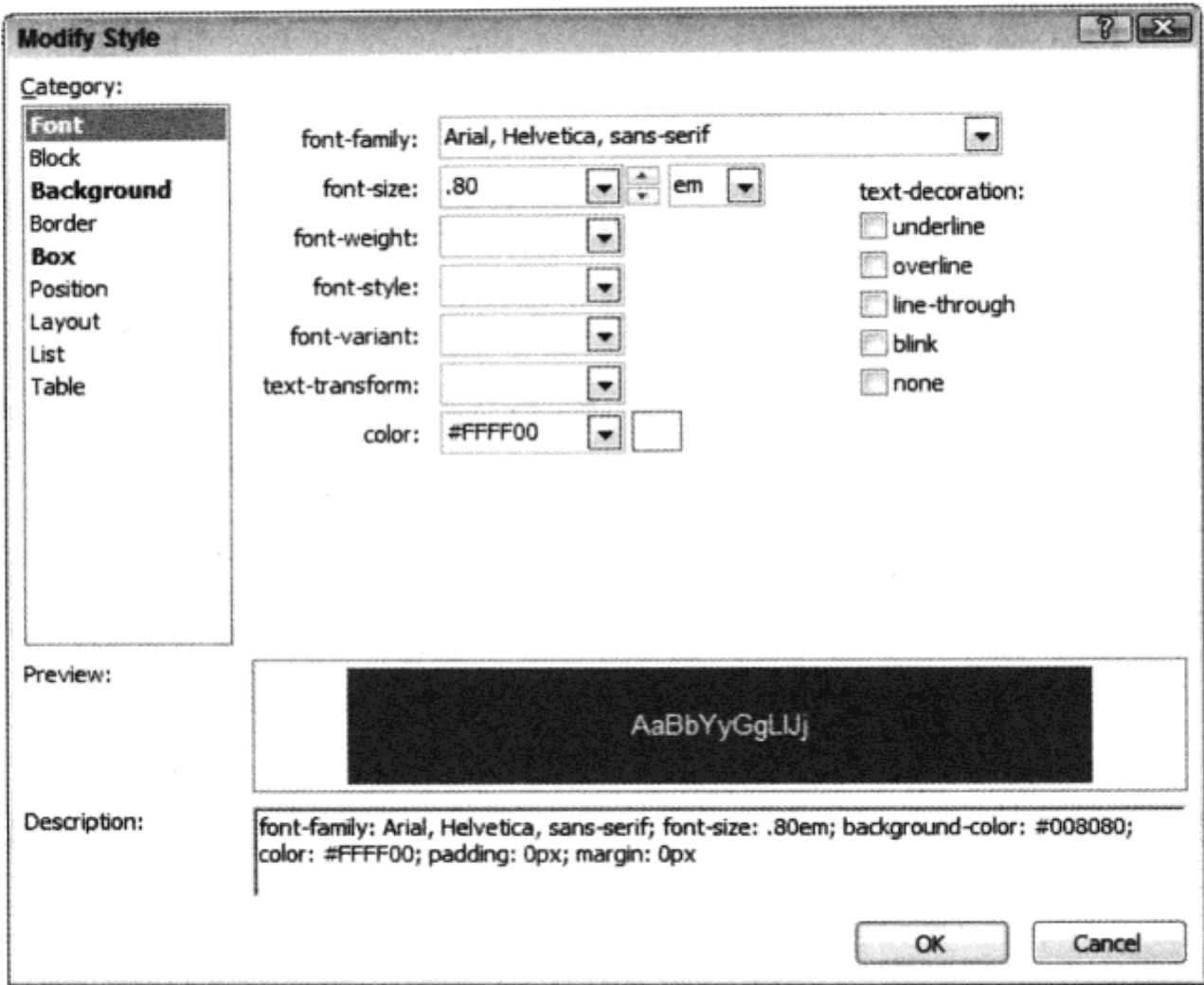



图 18-16

- (6) 选择 Font 类别，把 font-family 设置改为 Arial, Helvetica, sans-serif，把 font-size 设置改为.80 em，把 color 改为#FFFF00。
- (7) 在同一个对话框中选择 Background 类别，把背景色改为#008080。
- (8) 选择 Box 类别，把 padding 和 margin 都改为 0。
- (9) 样式表现在应如下面的代码段所示：



可从  
wrox.com  
下载源代码

```
body
{
    font-family: Arial, Helvetica, sans-serif;
    font-size: .80em;
    background-color: #008080;
    color: #FFFF00;
    padding: 0px;
    margin: 0px;
}
```

代码段 Site.css

- (10) 在 CSS 编辑器的源代码视图中，选择关联菜单 Add Style Rule，再选择元素 a:hover，如图 18-17 所示，单击 OK 按钮。

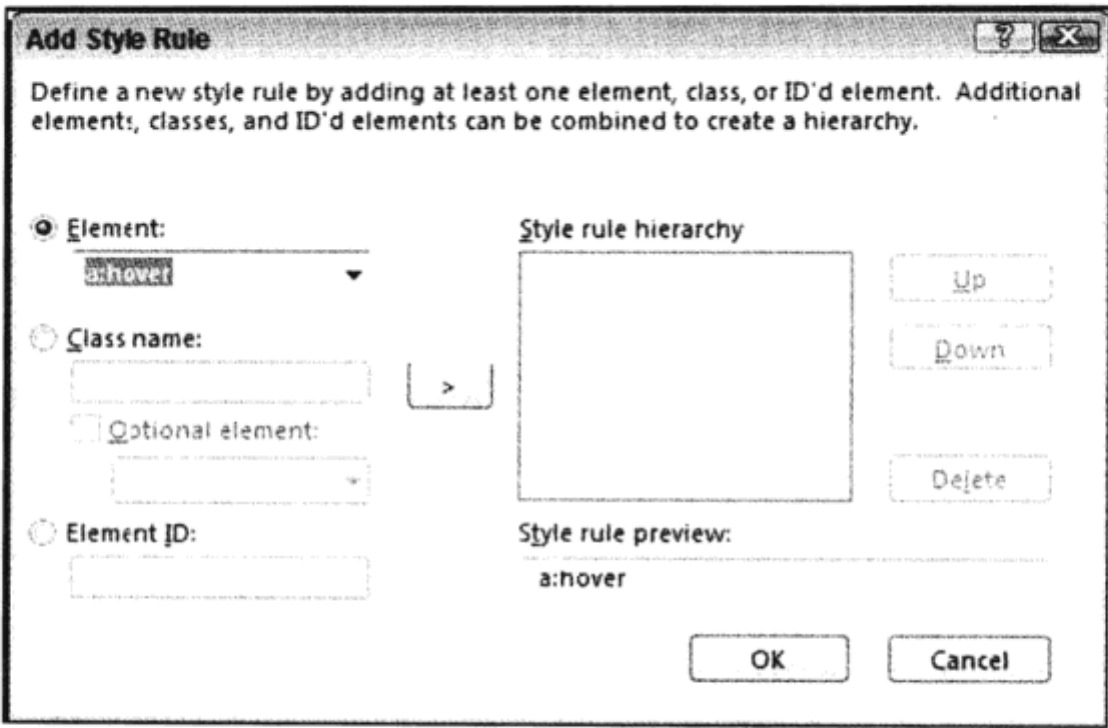


图 18-17

(11) 选择 Build Style 菜单，再次打开 Modify Style 对话框。选择 Font 类别，把颜色改为#FF0000，选择 text-decoration 中的 underline 和 overline，得到的 CSS 代码如下所示。



可从  
wrox.com  
下载源代码

```
a: hover
{
    color: #FF0000;
    text-decoration: underline overline;
}
```

代码段 Site.css

(12) 根据下面的代码给 a:active、a:link、a:visited 和 h1 添加样式规则：



可从  
wrox.com  
下载源代码

```
a: link, a: visited
{
    color: #00FFFF;
}

a: active
{
    color: #00FFFF;
}

a: hover
{
    color: #FF0000;
    text-decoration: underline overline;
}

h1
{
    text-align: center;
}
```

代码段 Site.css

- (13) 创建一个新的 Web 页面 StylesDemo.aspx，把 Site.css 文件从 Solution Explorer 拖放到编辑器的设计视图上。页面的背景色会立即改变。
- (14) 切换到源代码视图，验证新的 link 项引用了样式表。



```
<head runat="server">
  <title></title>
  <link href="Styles/Site.css" rel="stylesheet" type="text/css" />
</head>
```

代码段 StylesDemo.aspx

- (15) 在页面的 body 元素内部添加一个 h1 标记和一个锚定标记，如下所示：



```
<body>
  <form id="form1" runat="server">
    <h1>
      Styles Demo</h1>
    <div>
      <a href="http://www.wrox.com">Wrox Press</a>
    </div>
  </form>
</body>
```

代码段 StylesDemo.aspx

- (16) 选择 Debug | Start without Debugging，启动浏览器，来查看该页面。页面应如图 18-18 所示。验证是否把页面样式应用于标题和链接。把鼠标停放在链接上，查看颜色的变化和文本装饰。

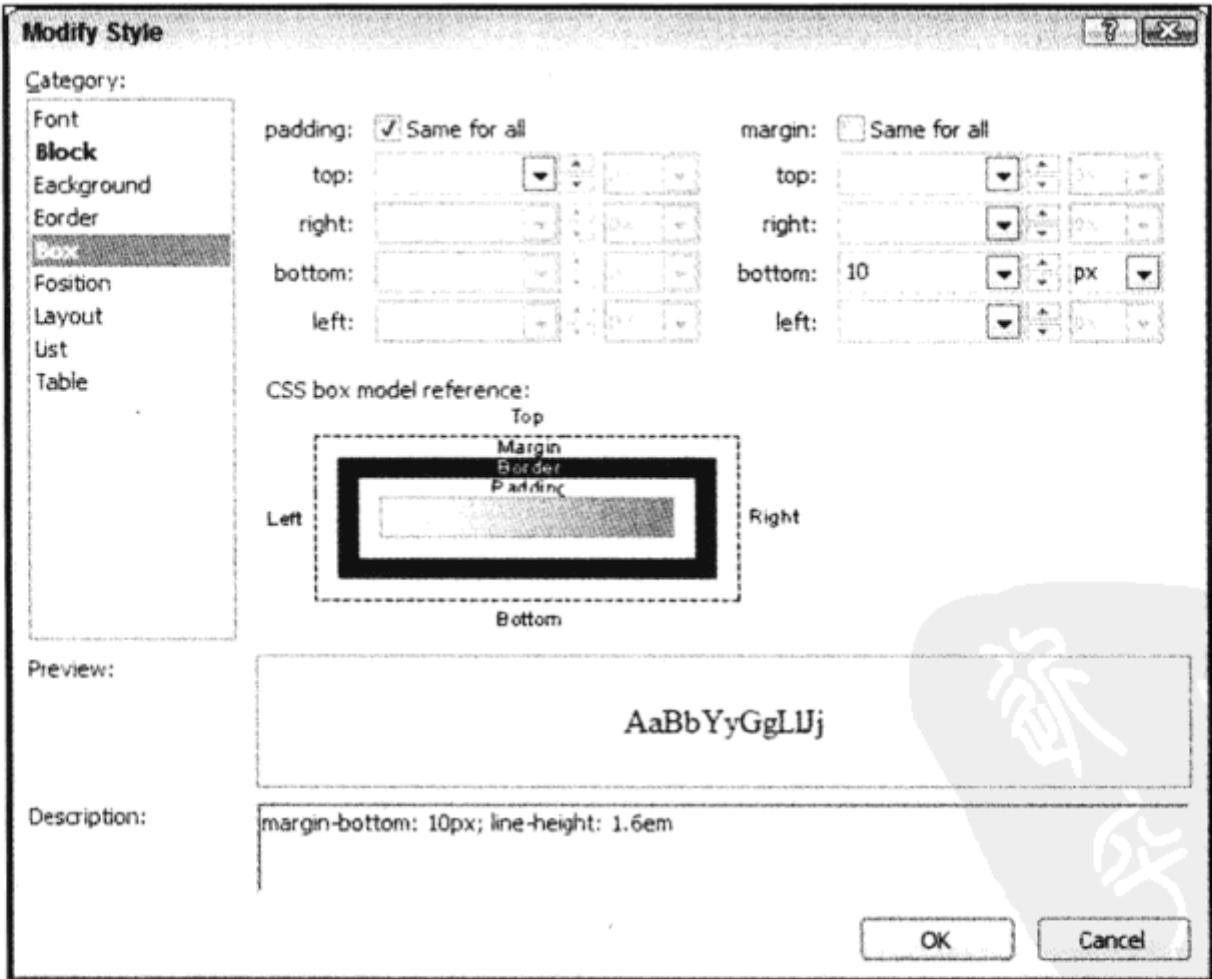


图 18-18

### 示例的说明

由于给链接元素引用了 CSS 文件，所以浏览器请求这个页面和 HTML 代码。接着浏览器使用 CSS 文件中已应用了样式的元素，来改变 HTML 元素的外观。

使用 CSS 不仅可以改变特定 HTML 标记的样式，还可以定义在 HTML 标记上引用的类，如下面的示例所示。

#### 试一试：定义样式类

- (1) 打开样式表 Site.css。
- (2) 打开 Add Style Rule 编辑器，添加一个新类 bottom。对话框中的样式规则预览在类名之前加了一个句点(.)。单击 OK 按钮。
- (3) 打开 Modify Style 编辑器。
- (4) 选择 Font 类别，给 font-size 选择 x-small。
- (5) 选择 Block 类别，将 vertical-align 定义为 text-bottom，将 text-align 定义为 center。
- (6) 选择 Box 类别，将上、下、左、右的页边距都定义 5。
- (7) 选择 Position 类别，把高度定义为 40 px。
- (8) 在 Site.css 文件中验证结果。



```
.bottom
{
    margin: 5px;
    height: 40px;
    text-align: center;
    vertical-align: text-bottom;
    font-size: x-small;
}
```

代码段 Site.css

- (9) 打开 StylesDemo.aspx 文件，添加一个包含如下文本的 div 元素：



```
<div class="bottom">
    Copyright (c) 2010 Wrox Press
</div>
```

代码段 StylesDemo.aspx

- (10) 启动浏览器，查看 StylesDemo.aspx 文件。验证给 div 元素应用了样式。

### 示例的说明

不需要为网站的每个页面中的每个元素定义样式，而可以在一个公共位置定义样式。Modify Style 编辑器可以预览所有可以改变的样式。打开页面时，浏览器会自动应用样式，并排列元素。



**警告：**在不同的浏览器上，一些样式的效果会不同。确保在所有支持的浏览器上查看页面的外观。



使用样式不仅可以应用字体大小和颜色,还可以定义 Web 页面的布局。另外,不仅可以利用样式来指定布局,还可以用 HTML 表指定布局。CSS 在布局方面比 HTML 表更灵活,而且还把内容与可见的信息分开,提供了可访问性方面的优势。因此目前无表格的 Web 设计通常是首选方案。

由于本书介绍 C# 编程,而不是 HTML 页面的设计,所以仅简要介绍了 CSS。如需了解 CSS 的更多信息,请参阅 Jon Duckett 编著的图书 *Beginning HTML, XHTML, CSS, and JavaScript* (Wrox 于 2009 出版)。

## 18.10 母版页

大多数 Web 站点都在每个页面上重用了母版页的部分内容,例如,公司徽标和菜单常常会出现在所有的页面上。每个页面不需要重复共同的用户界面元素,可以将共同的元素添加到母版页上。母版页看起来类似于一般的 ASP.NET 页面,但定义了由内容页(content page)替换的占位符。

母版页的文件扩展名是 .master,它在文件的第一行上使用 Master 指令,如下所示:

```
<%@ MasterLanguage="C#" AutoEventWireup="true" CodeFile="MasterPage.master.cs"
    Inherits="MasterPage" %>
```

在 Web 站点中,只有母版页才使用 <html>、<head>、<body> 和 <form> 等 HTML 元素。Web 页面本身只包含内嵌到 <form> 元素中的内容。Web 页面可以把它自己的内容嵌入 ContentPlaceHolder 控件中。如果 Web 页面没有这么做,母版页可以为 ContentPlaceHolder 定义默认内容。

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> </title>
    <asp:contentplaceholder id="head" runat="server">
    </asp:contentplaceholder>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:contentplaceholder id="ContentPlaceHolder1" runat="server">
            </asp:contentplaceholder>
        </div>
    </form>
</body>
</html>
```

要使用母版页,必须把 MasterPageFile 特性应用于 Page 指令。要替换母版页的内容,可以使用 Content 控件。该控件把 ContentPlaceHolder 与 ContentPlaceHolderID 关联起来。

```
<%Page Language="C#" MasterPageFile="~/MasterPage.master"
    AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="default"
    Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head">
```

```

    Runat="Server"></asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
    Runat="Server"></asp:Content>

```

除了用 Page 指令定义母版页之外，还可以在 Web 配置文件 web.config 中使用<pages>元素，把默认的母版页赋予所有 Web 页面：

```

<configuration>
  <system.web>
    <pages masterPageFile="~/MasterPage.master" />
    <!--...-->
  </pages>
</system.web>
</configuration>

```

使用在 web.config 中配置的母版页文件时，ASP.NET 页面需要该文件中的 Content 元素配置，如前所示。否则，masterPageFile 特性就没有用了。如果同时使用 Page 指令的 MasterPageFile 特性和 web.config 中的项，Page 指令的设置将覆盖 web.config 中的设置。这样，就可以既使用 web.config 定义默认的母版页文件，又允许特定的 Web 页面覆盖默认设置。

还可以使用编程方式修改母版页。这样就可以把不同的母版页用于不同的设备或不同的浏览器类型。可以修改母版页的最后一个地方是在 Page\_PreInit 处理方法中。在下面的示例代码中，如果浏览器发送 MSIE 字符串和浏览器名称(由 Microsoft IE 发送)，则 Page 类的 MasterPageFile 属性就设置为 IE.master，否则给其他所有浏览器发送 Default.master：

```

public partial class changeMaster : System.Web.UI.Page
{
    void Page_Load(object sender, EventArgs e)
    {
    }

    void Page_PreInit(object sender, EventArgs e)
    {
        if (Request.UserAgent.Contains("MSIE"))
        {
            this.MasterPageFile = "~/IE.master";
        }
        else
        {
            this.MasterPageFile = "~/Default.master";
        }
    }
}

```

在下面的示例中将创建自己的母版页。这里的示例母版页包含标题和主体，母版页的主要部分由各个页面替代。

### 试一试：创建母版页

- (1) 打开 Web 应用程序项目 EventRegistrationWeb。
- (2) 添加一个新的 Master Page 项，命名为 Events.master。



(3) 切换到编辑器的设计视图，应用样式表 Site.css。把 Site.css 文件从 Solution Explorer 拖放到编辑器中。

(4) 把第二个 ContentPlaceHolder 的 ID 重命名为 ContentPlaceHolderMain，把 CSS 类的内容赋予 div 元素，如下所示：



可从  
wrox.com  
下载源代码

```
<div class="content">
    <asp:ContentPlaceHolder ID="ContentPlaceHolderMain" runat="server">
    </asp:ContentPlaceHolder>
</div>
```

代码段 Events.Master

(5) 在前面修改的 div 元素前面，添加下面的 div 和 h1 元素：



可从  
wrox.com  
下载源代码

```
<div class="header">
    <h1>
        Event Registration
    </h1>
</div>
<div class="navigation">
    Menu will go here
</div>
```

代码段 Events.Master

(6) 在包含 ContentPlaceHolder 的 div 元素后面添加如下 div 元素：



可从  
wrox.com  
下载源代码

```
<div class="bottom">
    Copyright (c) 2010 Wrox Press
</div>
```

代码段 Events.Master

(7) 完整的页面如下所示：



可从  
wrox.com  
下载源代码

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Events.master.cs"
    Inherits="EventRegistrationWeb.Events" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <asp:ContentPlaceHolder ID="head" runat="server">
    </asp:ContentPlaceHolder>
    <link href="Styles/Site.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
        <div class="header">
            <h1>
                Event Registration
```

```
</h1>
</div>
<div class="navigation">
    Menu will go here
</div>
<div class="content">
    <asp:ContentPlaceHolder ID="ContentPlaceHolderMain" runat="server">
    </asp:ContentPlaceHolder>
</div>
<div class="bottom">
    Copyright (c) 2010 Wrox Press
</div>
</form>
</body>
</html>
```

代码段 Events.Master

示例的说明

如前所述，母版页包含 HTML，其中的 FORM 标记包括内容占位符，其内容会被使用母版页的页面替代。HTML 和链接的 CSS 定义了页面的布局。只有内容占位符会被内容页面替代。如果应替换页面的不同部分，就可以使用多个内容占位符。

创建母版页后，就可以在 Web 页面中使用它，如下面的示例所示。

试一试：使用母版页

- (1) 在 Web 应用程序中使用母版页添加一个 Web Form 类型的新项，命名为 Default.aspx。
- (2) 打开 Select a Master Page 对话框，如图 18-19 所示。选择母版页 Events.master，单击 OK 按钮。

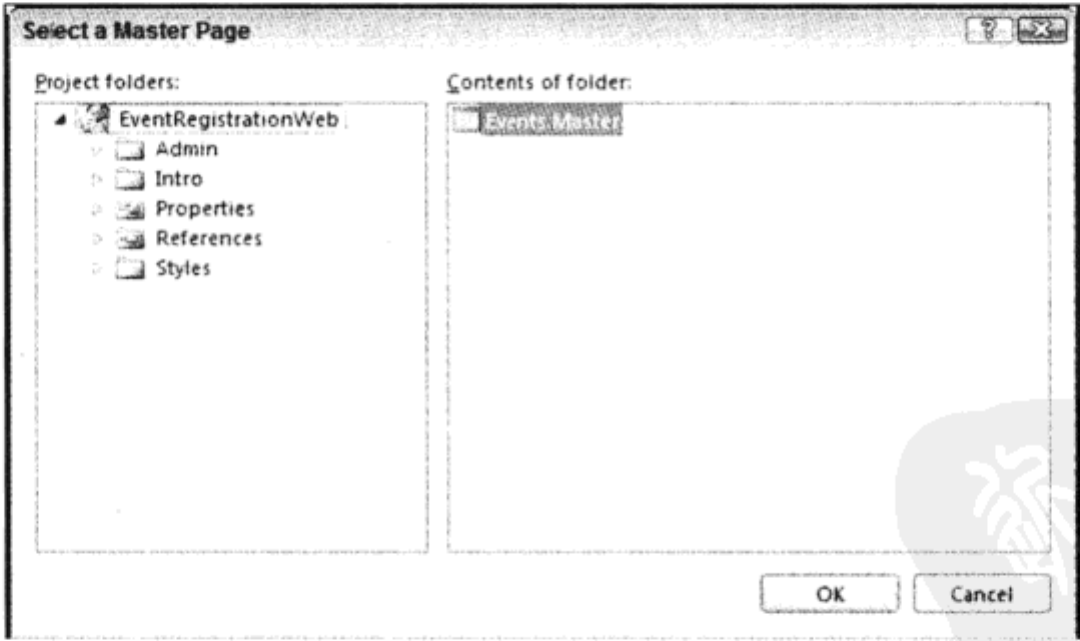


图 18-19

- (3) 在 Default.aspx 文件的源代码视图中，Page 指令引用了母版页中的 ContentPlaceHolder 控件，在该指令后只显示了 2 个 Content 控件。把这两个 Content 控件的 ID 属性改成 ContentHead 和 ContentMain。



可从  
wrox.com  
下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.master"
    AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="EventRegistrationWeb.Default" %>
<asp:Content ID="ContentHead" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="ContentMain" ContentPlaceHolderID="ContentPlaceHolderMain"
    runat="server">
</asp:Content>
```

代码段 Default.aspx

(4) 切换到 Visual Studio 的设计视图。这个视图显示了不能在页面中修改的母版页内容, 包括标题和版权信息, 给 Content 控件添加一些文本, 并居中对齐。

(5) 切换到源代码视图, 会看到如下代码。居中对齐改为 CSS 样式。



可从  
wrox.com  
下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
    AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="EventRegistrationWeb.Default" %>
<asp:Content ID="ContentHead" ContentPlaceHolderID="head" runat="server">
    <style type="text/css">
        .style1
        {
            text-align: center;
        }
    </style>
</asp:Content>
<asp:Content ID="ContentMain" ContentPlaceHolderID="ContentPlaceHolderMain"
    runat="server">
    <p class="style1">
        Welcome to the</p>
    <p class="style1">
        Event Registration</p>
    <p class="style1">
        Sample application for Beginning Visual C# 2010!</p>
</asp:Content>
```

代码段 Default.aspx

(6) 用浏览器查看页面。结果应如图 18-20 所示。

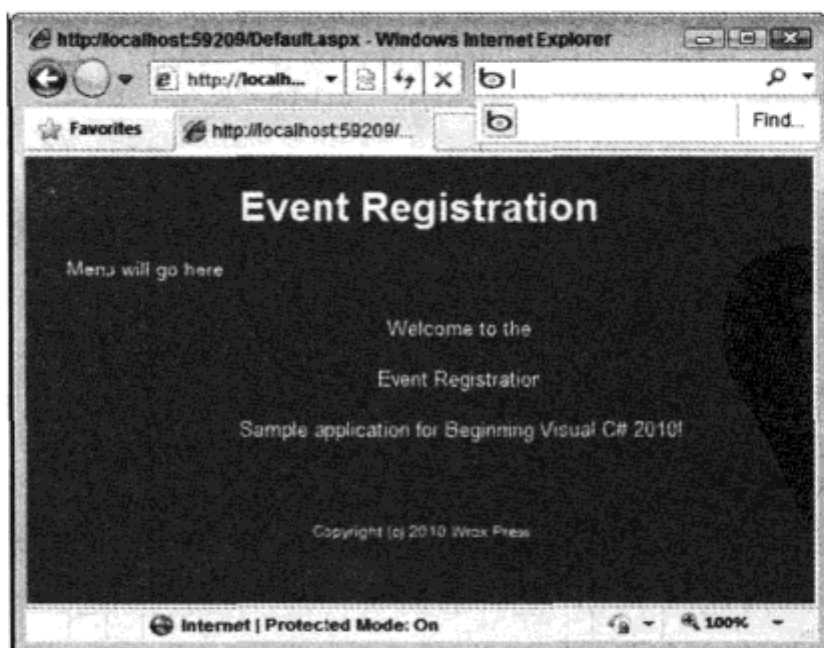


图 18-20

18.11 站点导航

在 Web 站点的多个页面上导航时，可以定义一个包含 Web 站点结构的 XML 文件，使用一些 UI 控件显示导航选项。用于导航的重要控件如表 18-8 所示。

表 18-8

控 件	说 明
SiteMapDataSource	SiteMapDataSource 控件是一个数据源控件，它引用站点地图数据提供程序。在 Visual Studio 工具箱中，该控件位于 Data 部分
Menu	Menu 控件按照站点地图数据源的定义显示页面的链接。菜单可以水平或垂直显示，而且有许多配置其样式的选项
SiteMapPath	SiteMapPath 控件使用很小的空间显示页面在 Web 站点层次结构中的当前位置。可以显示文本或图像超链接
TreeView	TreeView 控件显示 Web 站点的层次结构视图

下面的示例添加一个站点地图和一个菜单控件，用于在网站的页面之间导航。

试一试：添加导航控件

- (1) 打开 Web 应用程序项目 EventRegistrationWeb。
- (2) 在 Solution Explorer 中右击项目，选择 Add New Item，给 Web 站点添加一个新的 Site Map 项，名称 Web.sitemap 不变。
- (3) 修改文件的内容，如下所示：



```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="Default.aspx" title="Home">
    <siteMapNode url="EventRegister.aspx" title="Register"
      description="Register to an Event" />
    <siteMapNode url="EventList.aspx" title="Event List"
      description="Lists Events Worldwide" />
    <siteMapNode url="Admin/EventManager.aspx" title="Event Management"
      description="Management of Events" roles="Editors" />
  </siteMapNode>
</siteMap>
```

代码段 Web.sitemap

- (4) 打开 Events.master 文件。
- (5) 在工具箱的 Data 选项卡上，找到 SiteMapDataSource 控件，把它添加到页面上。
- (6) 从工具箱的 Navigation 选项卡上，把一个 Menu 控件添加到标题 Registration Demo Web 的下面。把数据源设置为 SiteMapDataSource1。
- (7) 配置 Menu 控件：把 Orientation 属性设置为 Horizontal，StaticDisplayLevels 属性设置为 2，

CssClass 属性设置为 menu。



可从  
wrox.com  
下载源代码

```
<div class="navigation">
    &nbsp;
    <asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1"
        Orientation="Horizontal" StaticDisplayLevels="2" CssClass="menu">
    </asp:Menu>
    <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</div>
```

代码段 Events.Master

(8) 在 Site.css 文件中添加如下样式规则，给菜单指定样式：



可从  
wrox.com  
下载源代码

```
.menu ul li a
{
    background-color: #008085;
    border: 1px #4e667d solid;
    color: #dde4ec;
    display: block;
    line-height: 1.35em;
    padding: 4px 20px;
    text-decoration: none;
    white-space: nowrap;
}

.menu ul li a:hover
{
    background-color: #bfc6d6;
    color: #465c71;
    text-decoration: none;
}
```

代码段 Site.css

(9) 在 Menu 控件的下面添加一个 SiteMapPath 控件。

(10) 在浏览器中打开 Default.aspx 文件。注意，菜单和路径显示了当前文件在 Web 站点中的位置。

(11) 使用 Web Form using Master Page 模板创建新页面 EventRegister.aspx 和 EventList.aspx，其中使用了母版页 Events.Master。

(12) 创建一个新文件夹 Admin，再在这个文件夹中创建一个新页面 EventManagement.aspx。再次使用 Web Form using Master Page 模板创建该页面。

(13) 根据需要添加在 Web.sitemap 文件中引用的其他页面。只需引用同一个母版页，显示定义好的菜单即可。

(14) 在 system.web 元素中，给 web.config 文件添加 siteMap 元素，如下所示：



可从  
wrox.com  
下载源代码

```
<siteMap defaultProvider="XmlSiteMapProvider" enabled="true">
    <providers>
        <clear />
        <add name="XmlSiteMapProvider"
            description="Default SiteMap Provider"
```

```
type="System.Web.XmlSiteMapProvider"
siteMapFile="Web.sitemap"
securityTrimmingEnabled="true" />
</providers>
</siteMap>
```

代码段 Web.config

### 示例的说明

Web 站点的结构由 Web.sitemap 文件中的 Web 页面定义。这个 XML 文件在<sitemap>根元素中包含 XML 元素<siteMapNode>。<siteMapNode>元素定义了 Web 页面。页面的文件名用 url 特性设置, title 特性指定显示在菜单中的名称。把<siteMapNode>元素编写为页面的子元素(包含子元素的链接), 就定义了页面的层次结构。

SiteMapDataSource 控件是一个数据源控件, 类似于第 17 章介绍的数据源控件。这个控件可以使用不同的提供程序。默认使用 XmlSiteMapProvider 类来获取数据。XmlSiteMapProvider 类默认使用 Web.sitemap 文件。

因为 roles 特性应用于 siteMapNode EventManagement.aspx, 所以只有特定角色 Editors 中的用户才能看到这个菜单项。因为 XmlSiteMapProvider 的这个授权特性默认情况下不启用, 所以把 web.config 文件改为设置 XmlSiteMapProvider 的 securityTrimmingEnabled 属性。如果菜单不需要角色, 就根本不需要 web.config 中的这个配置。

使用 Menu 控件可以编辑显示在 ASPX 源文件中的菜单项, 也可以通过编程方式添加菜单项。添加菜单项最简单的方式是通过配置数据源来使用站点地图数据源。

## 18.12 身份验证和授权

为了保护网站, 应进行身份验证, 检查用户是否有权登录。而授权检查的是已通过身份验证的用户是否可以使用资源。

ASP.NET 提供了 Windows 和 Forms 身份验证。对于 Web 应用程序, 最常用的身份验证技术是 Forms 身份验证。ASP.NET 还为 Forms 身份验证提供了一些新特性。Windows 身份验证利用 Windows 账户和 IIS 来验证用户的身份。

ASP.NET 为用户的身份验证提供了许多类。图 18-21 列出了新体系结构。在 ASP.NET 中, 有许多新的安全控件, 如 Login 和 PasswordRecovery。这些控件都利用了 Membership API。在 Membership API 中, 可以创建和删除用户、验证登录信息或获取当前登录的用户信息。Membership API 本身又利用了成员提供程序。在 ASP.NET 4 中, 存在不同的提供程序, 用于访问 Access 数据库、SQL Server 数据库或 Active Directory 中的用户。还可以创建定制的提供程序, 来访问 XML 文件或其他定制存储器。

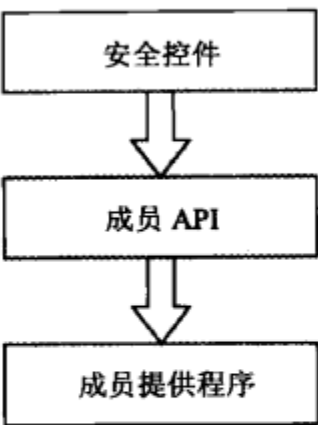


图 18-21

18.12.1 身份验证的配置

本章将演示 Forms 身份验证和一个成员提供程序。下面的示例会为 Web 应用程序配置安全性，给不同的文件夹赋予不同的访问列表。

试一试：安全配置

- (1) 用 Visual Studio 打开前面创建的 Web 应用程序 EventRegistrationWeb。
- (2) 在 Solution Explorer 中选择 Web 目录，再选择菜单项 Website | Add Folder | Regular Folder，创建一个新文件夹，把它命名为 Intro。这个文件夹要配置为由所有的用户访问，而主文件夹只能由授权的用户访问。前面创建的文件夹 Admin 只能由 Editors 角色中的用户访问。
- (3) 选择 Visual Studio 2010 中的菜单项 Project | ASP.NET Configuration，启动 ASP.NET Web Application Administration。
- (4) 选择 Security 选项卡，如图 18-22 所示。

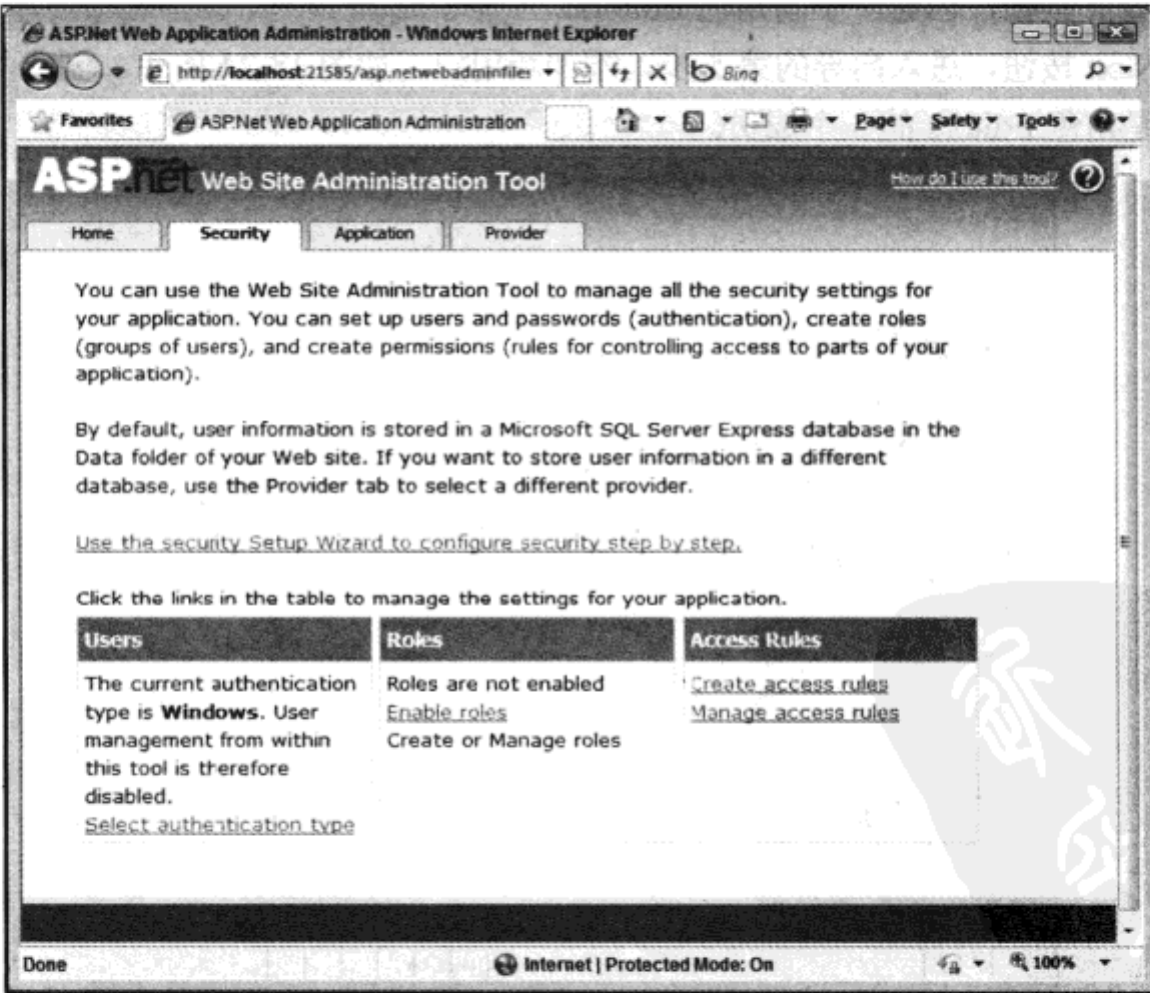


图 18-22



(5) 单击 Security Setup Wizard 链接。在欢迎屏幕中，单击 Next 按钮。在该向导的第 2 步中，选择访问方法 From the internet，如图 18-23 所示。

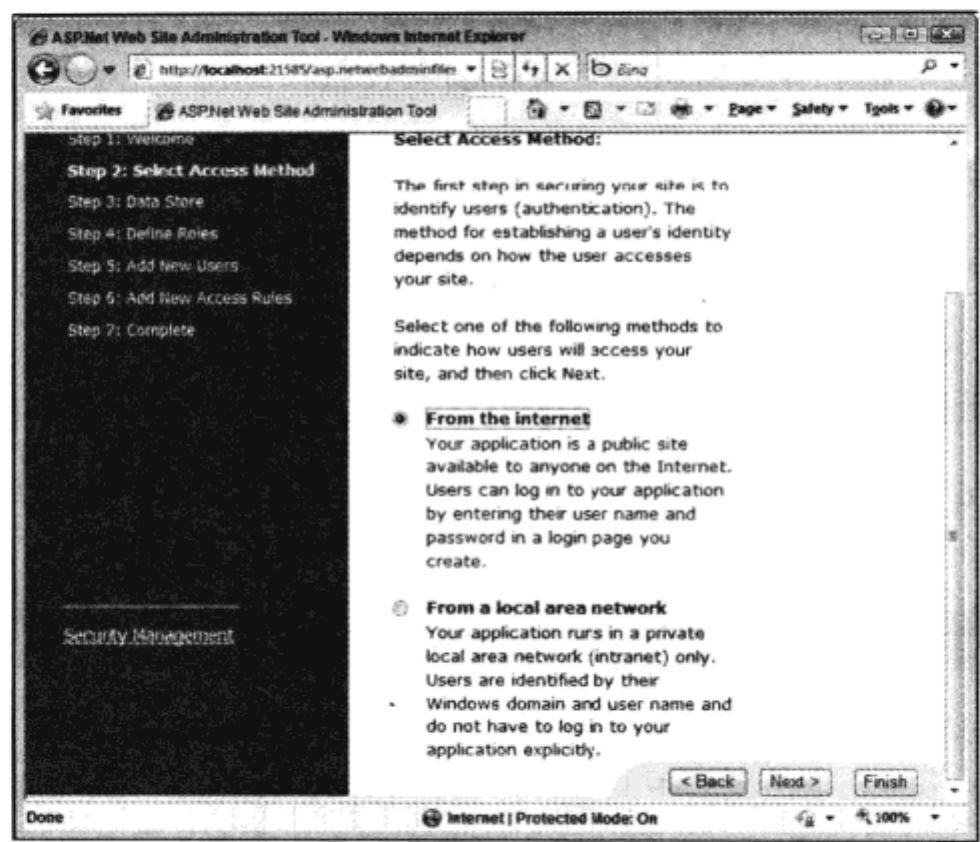


图 18-23

- (6) 单击 Next 按钮，进入向导的第 3 步，其中显示了已配置的提供程序信息。默认的提供程序是 SQL Server 数据库提供程序。这个配置不能在向导模式下修改，但可以在稍后进行修改。
- (7) 单击 Next 按钮，在 Define Roles 屏幕中，选中复选框 Enable roles for this Web site。
- (8) 单击 Next 按钮，创建一个新角色 Editors。
- (9) 单击 Next 按钮，进入向导的第 5 步，添加新用户，如图 18-24 所示。创建两个新账户。其中一个账户应是角色 Editors 的一个成员。

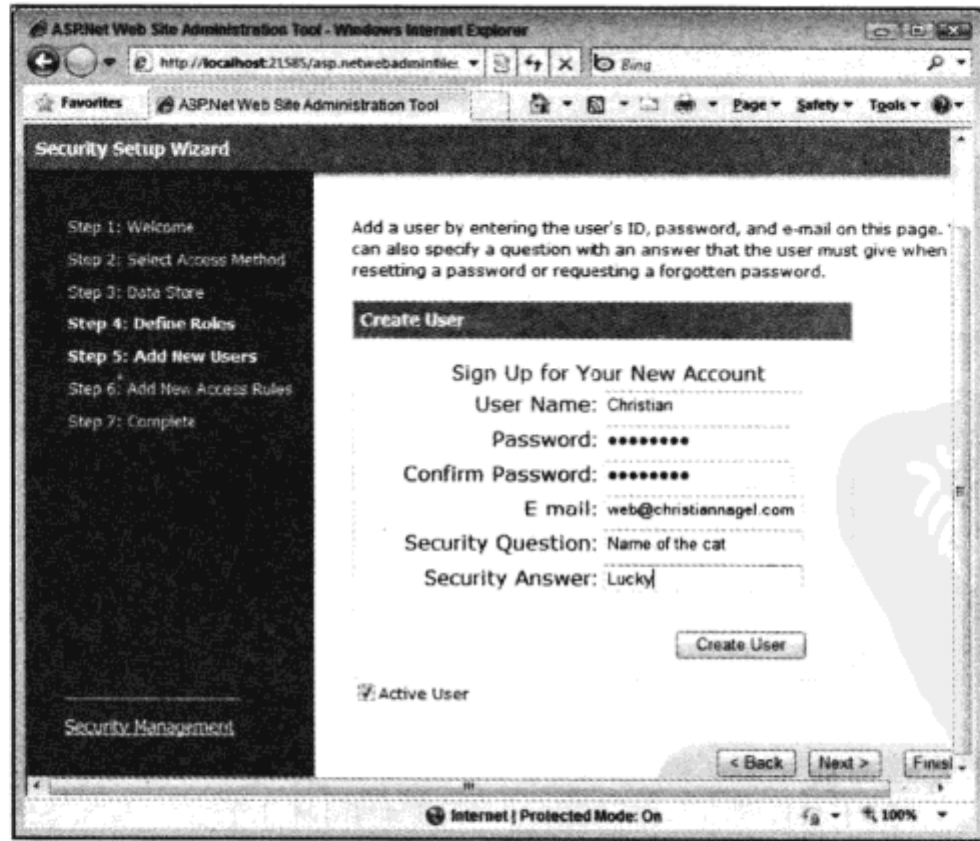


图 18-24

(10) 成功创建用户后，单击 **Next** 按钮，进入向导的第 6 步，如图 18-25 所示。在这里可以配置哪些用户可以访问网站或特定的目录。添加一个拒绝匿名用户的规则。接着选择目录 **Intro**，添加一个允许匿名用户访问的规则。选择 **Admin** 文件夹，拒绝通过身份验证的用户访问，但允许角色 **Editors** 的用户访问。之后单击 **Next** 按钮，最后单击 **Finish** 按钮。

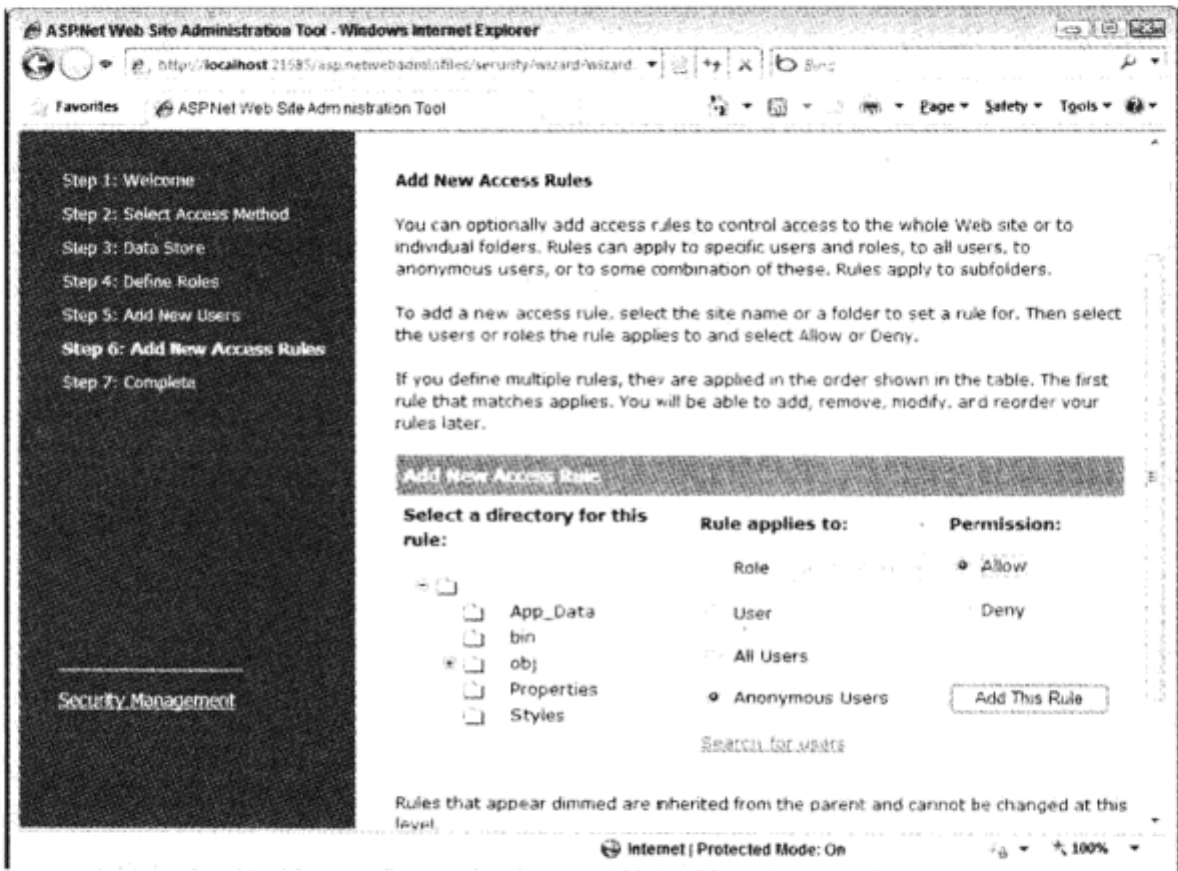


图 18-25

示例的说明

在完成安全配置后，就会创建一个新的 SQL Server 数据库。刷新 **Solution Explorer** 中的文件，就可以看到目录 **App\_Data** 下的新 SQL Server Express 数据库 **ASPNETDB.mdf**。这个数据库包含的表由 **SQL Membership** 提供程序使用。

现在，除了 Web 应用程序之外，还有一个配置文件 **web.config**。这个文件包含 **Forms** 身份验证的配置，因为选择了通过 **Internet** 进行身份验证，**<authorization>** 部分拒绝匿名用户的访问。如果 **Membership** 提供程序改变了，新的提供程序就会出现在配置文件中。**SQL** 提供程序是默认的，已经用计算机配置文件定义了，所以不需要列出来。



可从  
wrox.com  
下载源代码

```
<authorization>
  <deny users="?" />
</authorization>
<roleManager enabled="true" />
<authentication mode="Forms" />
```

代码段 Web.config

子文件夹 **Intro** 中包含另一个配置文件 **web.config**。在这个配置文件中没有身份验证部分，因为身份验证配置信息是从父目录中提取的。但是，授权部分是不同的，这里使用 **<allow users="?" />** 允许匿名用户的访问。



可从  
wrox.com  
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <allow users="?" />
    </authorization>
  </system.web>
</configuration>
```

代码段 Intro/Web.config

Admin 子文件夹中包含另一个配置文件 web.config。授权部分允许 Editors 角色的访问，拒绝通过身份验证的用户的访问：



可从  
wrox.com  
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Editors" />
      <deny users="*" />
    </authorization>
  </system.web>
</configuration>
```

代码段 Admin/Web.config

18.12.2 使用安全控件

ASP.NET 包含许多安全控件，现在不用编写一个定制的表单，让用户输入用户名和密码，而可以使用预定义的 Login 控件。安全控件及其功能如表 18-9 所示。

表 18-9

安全控件	说 明
Login	一个复合控件，包含要求用户输入用户名和密码的控件
LoginStatus	根据用户是否登录来决定包含登录或注销的超链接
LoginName	显示用户名
LoginView	根据用户是否登录显示不同的内容
PasswordRecovery	一个复合控件，用于重新设置被遗忘的密码。根据安全配置，要求用户回答以前设置的问题，或通过电子邮件发送密码
ChangePassword	一个复合控件，允许登录的用户改变其密码
CreateUserWizard	一个向导，可以创建新用户，把用户信息写入 Membership 提供程序

在下面的示例中，将给 Web 应用程序添加一个登录页面。

试一试：创建登录页面

如果在网站配置为拒绝匿名用户后启动它，就应接收到一个错误，因为该网站还没有 login.aspx

页面。如果没有用 Forms 身份验证配置某个登录页面，就使用默认的 login.aspx。下面创建一个 login.aspx 页面。

- (1) 使用母版页添加一个新的 Web Form，命名为 login.aspx。
- (2) 在窗体中添加 Login 控件。
- (3) 这就是创建登录页面所需要做的工作。现在启动站点 default.aspx，就会重定向到 login.aspx，在这个窗体中可以为前面创建的用户输入用户凭据。

#### 示例的说明

添加 Login 控件之后，就可以在源代码视图中查看到其代码：



可从  
wrox.com  
下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
    AutoEventWireup="true" CodeBehind="Login.aspx.cs"
    Inherits="EventRegistrationWeb.Login" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
    runat="server">
    <asp:Login ID="Login1" runat="server">
    </asp:Login>
</asp:Content>
```

代码段 Login.aspx

可以通过这个控件的属性来配置标题、用户名标签、密码标签和登录按钮的文本。设置 DisplayRememberMe 属性还可以使复选框 Remember me next time 可见。

如果要更多地控制 Login 控件的外观和操作方式，可以把该控件转换为模板。为此，在设计视图中单击智能标记，选择 Convert to Template。之后，单击 Edit Templates，就得到一个视图，在这里可以添加和修改任何控件。

对于用户凭据的验证，在单击 Login 按钮时，控件会调用 Membership.ValidateUser() 方法，不需要手工完成。

如果用户在 EventRegistration 网站上没有登录账户，就应创建自己的账户。为此可以使用 CreateUserWizard 控件，如下面的示例所示。

#### 试一试：使用 CreateUser 向导

- (1) 在前面创建的 Intro 文件夹中创建一个新的 Web 页面 RegisterUser.aspx，配置这个文件夹，使之可由匿名用户访问。
- (2) 在这个 Web 页面上添加一个 CreateUserWizard 控件。
- (3) 把 ContinueDestinationPageUrl 属性设置为 ~/Default.aspx。
- (4) 在 Login.aspx 页面上添加 LinkButton 控件。把这个控件的内容设置为 Register User，PostBackUrl 属性设置为 Web 页面 Intro/RegisterUser.aspx。
- (5) 启动应用程序。单击 Login.aspx 页面上的 Register User 链接，就会重定向到 RegisterUser.aspx 页面上，在这个页面上将用输入的数据创建一个新账户。

示例的说明

CreateUserWizard 控件类似于向导，包含多个向导步骤，它们是用<WizardSteps>元素定义的：



可从  
wrox.com  
下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
    AutoEventWireup="true" CodeBehind="RegisterUser.aspx.cs"
    Inherits="EventRegistrationWeb.Intro.RegisterUser" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
    runat="server">
    <asp:CreateUserWizard ID="CreateUserWizard1" runat="server">
        <WizardSteps>
            <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
                </asp:CreateUserWizardStep>
            <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
                </asp:CompleteWizardStep>
        </WizardSteps>
    </asp:CreateUserWizard>
</asp:Content>
```

代码段 RegisterUser.aspx

可以在设计器中配置这些向导步骤。控件的智能标记可以分别配置每个步骤。图 18-26 显示了步骤 Sign Up for Your New Account 的配置。也可以用定制控件添加定制步骤以便添加特定的要求，例如，用户应在签署一个账户前接受合同。

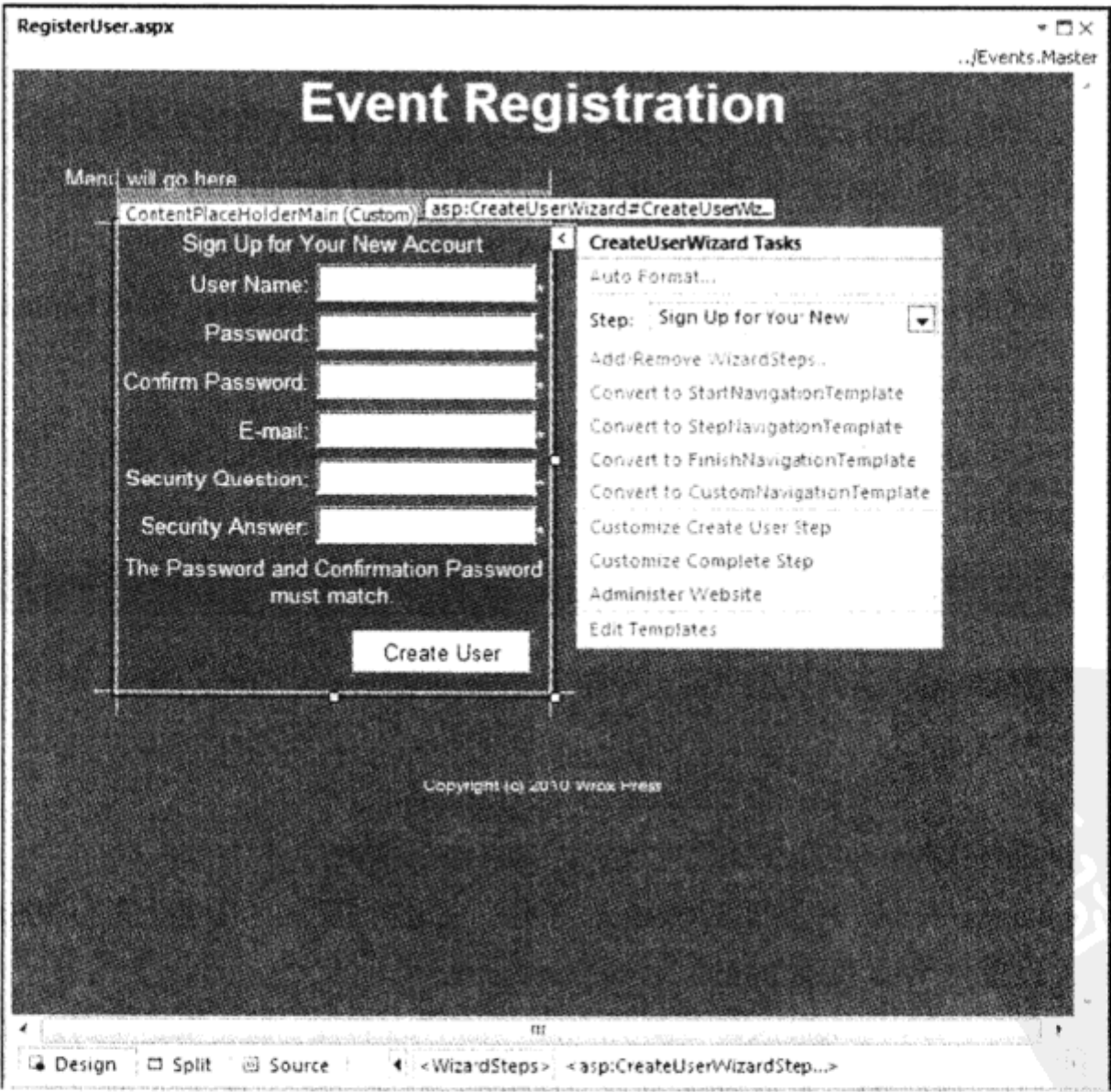


图 18-26

### 18.13 读写 SQL Server 数据库

大多数 Web 应用程序都需要访问数据库，在其中读写数据。本节将创建一个新数据库，以存储事件信息，说明如何在 ASP.NET 中使用这个数据库。首先在下面的示例中创建一个新的 SQL Server 数据库，这可以直接在 Visual Studio 2010 中完成。

**试一试：创建新数据库**

- (1) 打开前面创建的 Web 应用程序 EventRegistrationWeb。
- (2) 打开 Server Explorer。如果在 Visual Studio 中看不到它，可以使用菜单项 View | Other Windows | Server Explorer，打开该窗口。
- (3) 在 Server Explorer 中，选择 Data Connections，右击打开关联菜单，选择 Create New SQL Server Database，打开如图 18-27 所示的对话框。



图 18-27

- (4) 输入服务器名(local)\sqlexpress 和数据库名 BegVCSHarpEvents。
- (5) 创建数据库后，在 Server Explorer 中选择新数据库。
- (6) 在数据库下选择 Tables 项，再选择 Visual Studio 中的菜单项 Data | Add New | Table。
- (7) 现在输入列名和数据类型，如表 18-10 所示。

表 18-10

列 名	数 据 类 型
Id	int
Title	nvarchar(50)
Date	datetime
Location	nvarchar(50)



(8) 通过标识递增 1 和标识种子 1，把 ID 列配置为主键列。所有的列都配置为不允许使用 null 值。

(9) 保存该表，将其命名为 Events。

(10) 在表中添加一些事件，包含标题、日期和位置。

为了显示、编辑数据，工具箱提供了一个独立的 Data 区域，表示数据控件。数据控件可以分为两组：数据视图和数据源。数据源控件与数据源(如 XML 文件、SQL 数据库或.NET 类)关联在一起。数据视图与数据源相连接，就可以表示数据。表 18-11 中列出了所有数据控件。

表 18-11

数 据 控 件	说 明
GridView	用行和列来显示数据
DataList	在一列中显示所有的项
DetailsView	如果数据具有主从(master/detail)关系，DetailsView 控件就可以和 GridView 一起使用
FormView	显示数据源中的一行
Repeater	基于模板的控件，可用于定义从数据源中的数据生成什么 HTML 元素
ListView	基于模板，类似于 Repeater 控件

数据源控件及其功能如表 18-12 中所示。

表 18-12

数据源控件	说 明
SqlDataSource	访问 SQL Server 或其他 ADO.NET 提供程序(例如，Oracle、 ODBC 和 OLEDB)。在内部，它使用 DataSet 或 DataReader 类
AccessDataSource	可以使用 Access 数据库
EntityDataSource	.NET 4.0 中的新控件，允许把 ADO.NET Entity Framework 用作数据源
ObjectDataSource	允许把.NET 类用作数据源
XmlDataSource	允许访问 XML 文件。使用这个数据源可以显示层次结构
SiteMapDataSource	使用 XML 文件定义站点结构，创建到该网站的链接和引用。这个功能将在第 20 章讨论

下面的示例将使用 GridView 控件显示、编辑前面创建的数据库中的数据。

试一试：使用 GridView 控件显示数据

- (1) 在 Admin 文件夹中打开前面创建的 Web 页面 EventsManagement.aspx。
- (2) 在 Web 页面中添加一个 GridView 控件。
- (3) 在控件智能标记的 Choose Data Source 组合框中，选择<New data source.>，打开如图 18-28 所示的对话框。



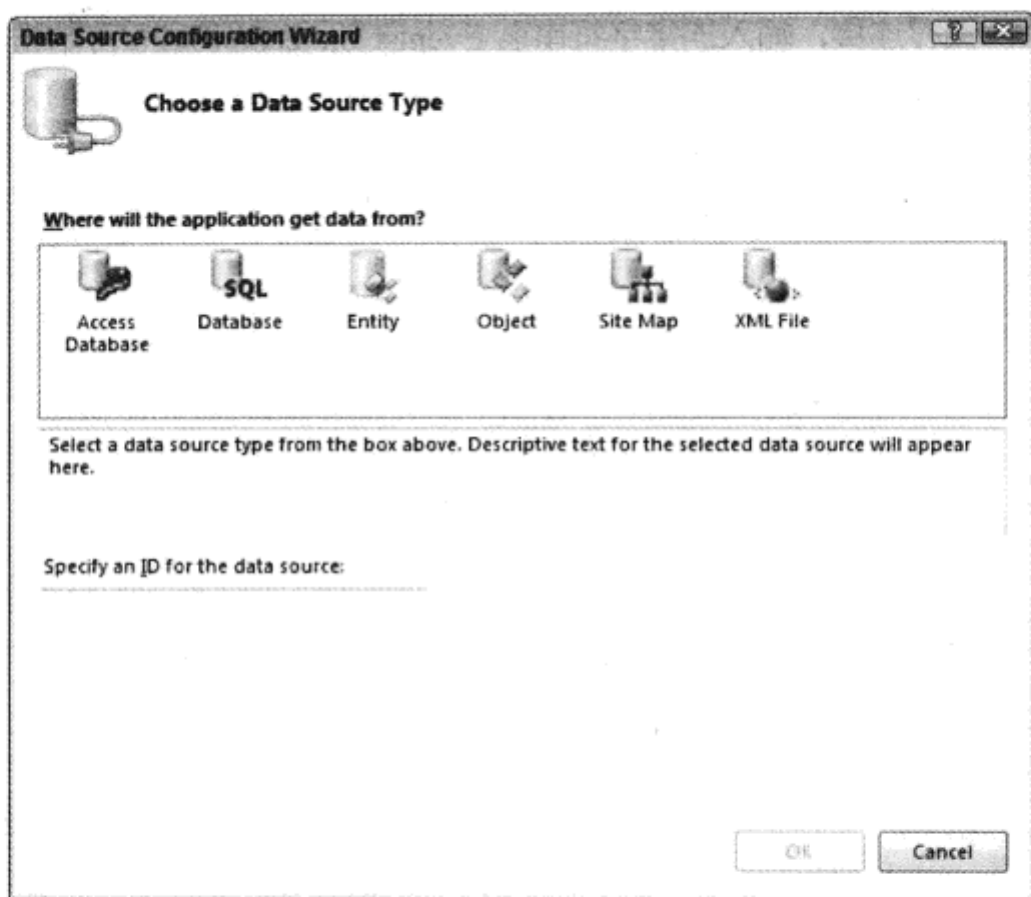


图 18-28

- (4) 选择 Database，为这个新数据源输入名称 EventsDataSource。
- (5) 单击 OK 按钮配置数据源，打开 Configure Data Source 对话框，单击 New Connection 按钮，创建一个新连接。
- (6) 在 Add Connection 对话框中，输入(local)\sqlexpress 作为服务器名，选择前面创建的数据库 BegVCSharpEvents。单击 Test Connection 按钮，验证是否正确配置了连接。满意后单击 OK 按钮。打开下一个对话框(用于存储连接字符串，如图 18-29 所示)。

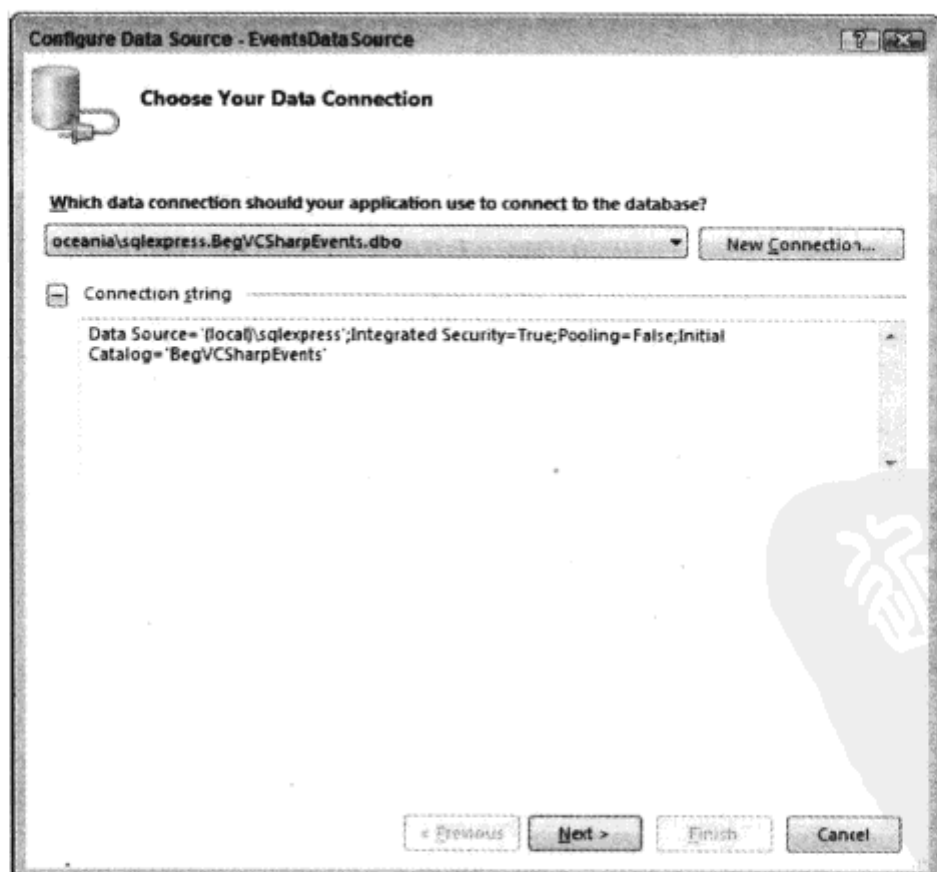


图 18-29

- (7) 选中复选框，保存连接，输入连接字符串名 EventsConnectionString，单击 Next 按钮。
- (8) 在下一个对话框中，选择 Events 表，以便从这个表中读取数据，如图 18-30 所示。选择 ID、Title、Date 和 Location 列，以定义图中显示的 SQL 命令。接着单击 Next 按钮。

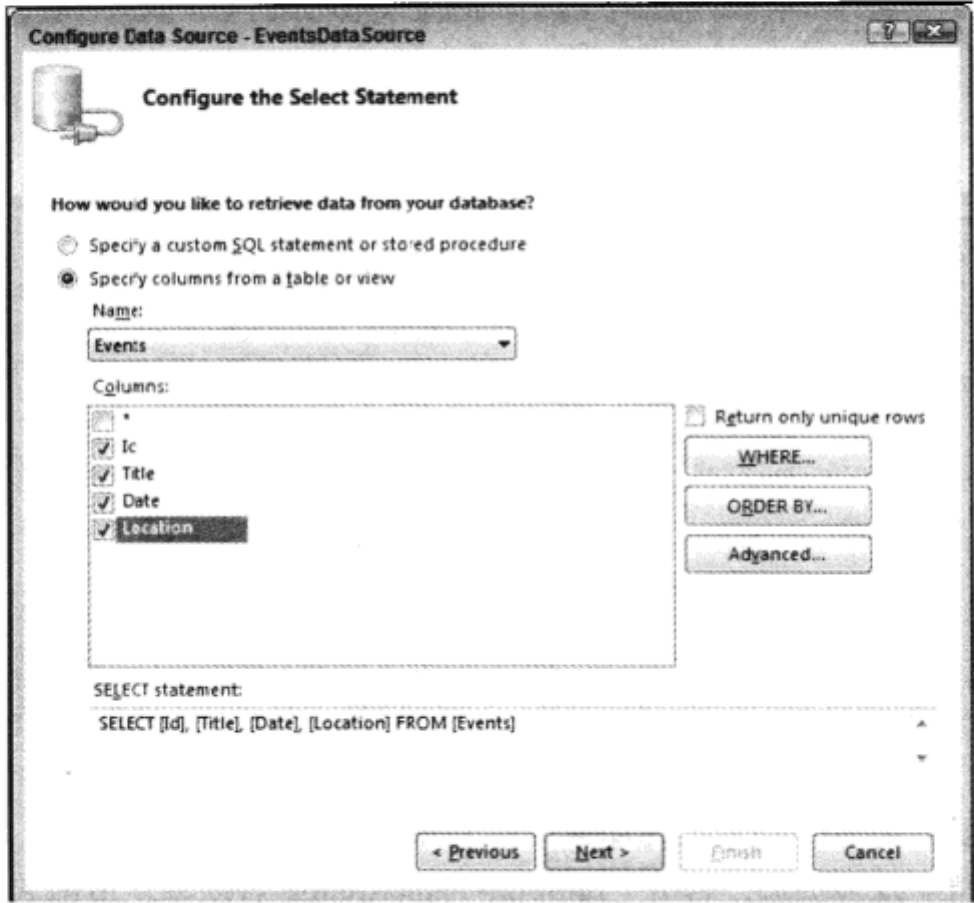


图 18-30

- (9) 在 Configure Data Source 对话框的最后一个窗口中，可以测试查询。最后单击 Finish 按钮。
- (10) 在设计器中，GridView 控件带有一些虚拟数据，SqlDataSource 的名称是 EventsDatasource。
- (11) 为了使 GridView 控件的布局更漂亮，从智能标记中选择 AutoFormat，再选择模式 Mocha，如图 18-31 所示。

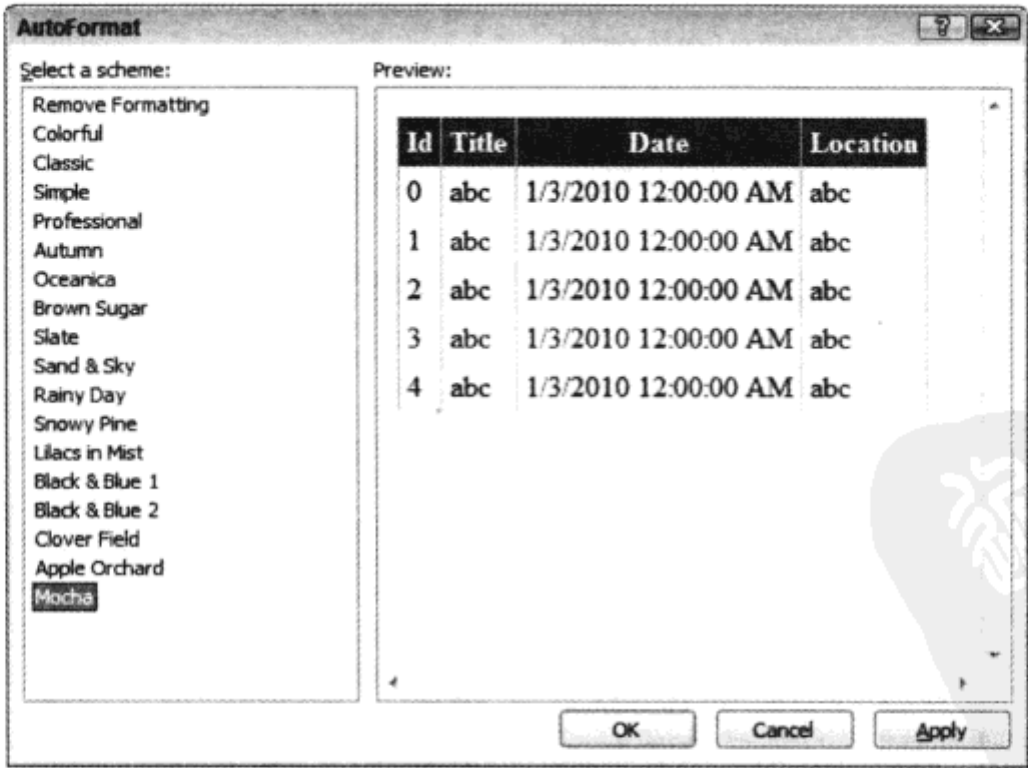


图 18-31

- (12) 在 Visual Studio 中启动页面，事件存储在一个美观的表格中，如图 18-32 所示。

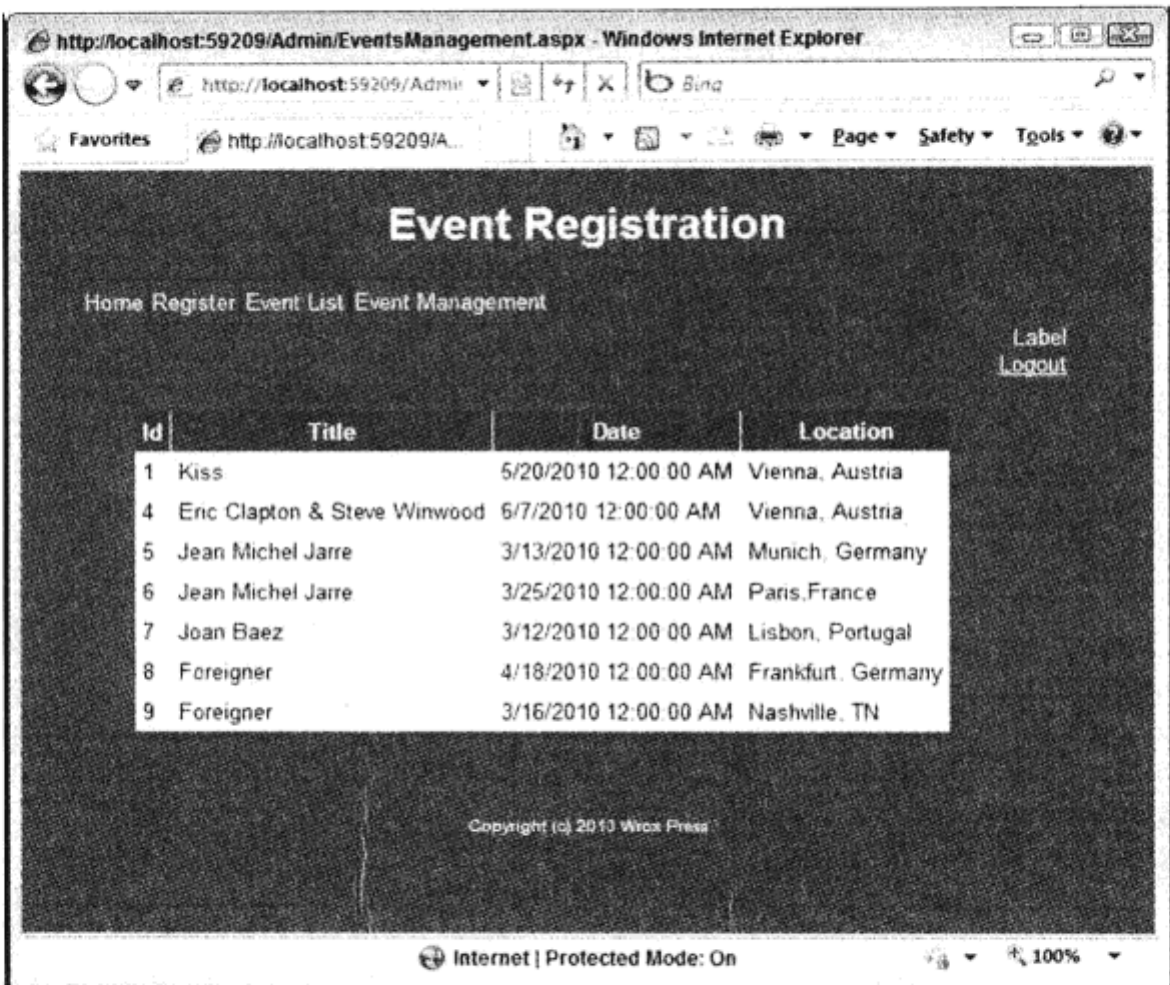


图 18-32

示例的说明

添加了 GridView 控件之后，就可以在源代码中看到它的配置。DataSourceID 特性定义了与数据源控件的关联，数据源控件在栅格控件的后面。在<Columns>元素中，列出了用于显示数据的所有绑定列。HeaderText 定义了标题的文本，DataField 定义了数据源中的字段名。

用<asp:SqlDataSource>元素定义数据源，其中 SelectCommand 定义了如何从数据库中读取数据，ConnectionString 定义了如何连接数据库。因为我们选择在配置文件中保存连接字符串，所以使用<%\$建立与配置文件中动态生成的类之间的关联。



可从  
wrox.com  
下载源代码

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
    AutoEventWireup="true" CodeBehind="EventsManagement.aspx.cs"
    Inherits="EventRegistrationWeb.Admin.EventsManagement" %>

<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
    runat="server">

    <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
        BackColor="White" BorderColor="#DEDFDE" BorderStyle="None"
        BorderWidth="1px" CellPadding="4" DataKeyNames="Id"
        DataSourceID="EventsDataSource" ForeColor="Black"
        GridLines="Vertical" PageSize="5">
        <AlternatingRowStyle BackColor="White" />
        <Columns>
            <asp:BoundField DataField="Id" HeaderText="Id" InsertVisible="False"
                ReadOnly="True" SortExpression="Id" />
            <asp:BoundField DataField="Title" HeaderText="Title"
                SortExpression="Title" />
            <asp:BoundField DataField="Date" HeaderText="Date"
```

```

        SortExpression="Date" />
    <asp:BoundField DataField="Location" HeaderText="Location"
        SortExpression="Location" />
</Columns>
<FooterStyle BackColor="#CCCC99" />
<HeaderStyle BackColor="#6B696B" Font-Bold="True" ForeColor="White" />
<PagerStyle BackColor="#F7F7DE" ForeColor="Black"
    HorizontalAlign="Right" />
<RowStyle BackColor="#F7F7DE" />
<SelectedRowStyle BackColor="#CE5D5A" Font-Bold="True"
    ForeColor="White" />
<SortedAscendingCellStyle BackColor="#FBFBF2" />
<SortedAscendingHeaderStyle BackColor="#848384" />
<SortedDescendingCellStyle BackColor="#EAEAD3" />
<SortedDescendingHeaderStyle BackColor="#575357" />
</asp:GridView>
<asp:SqlDataSource ID="EventsDataSource" runat="server"
    ConnectionString="<%= ConnectionStrings.BegVCSharpEventsConnectionString %>"
    SelectCommand="SELECT [Id], [Title], [Date], [Location] FROM [Events]">
</asp:SqlDataSource>
</asp:Content>

```

代码段 EventsManagement.aspx

可以在配置文件 web.config 中找到数据库的连接字符串：



可从  
wrox.com  
下载源代码

```

<connectionStrings>
  <add name="BegVCSharpEventsConnectionString"
    connectionString="Data Source='(local)\sqlexpress' ;
    Integrated Security=True;Pooling=False;
    Initial Catalog=' BegVCSharpEvents' "
    providerName="System.Data.SqlClient" />
</connectionStrings>

```

代码段 web.config

现在 GridView 控件的配置应有所不同。在下面的示例中，不再向用户显示 ID，只显示日期，不显示时间。

### 试一试：配置 GridView 控件

(1) 选择 GridView 控件的智能标记，再选择 Edit Columns 菜单。打开如图 18-33 所示的 Fields 对话框。选择 Id 字段，把 Visible 属性改为 False。也可以用这个对话框安排列的位置，改变颜色，定义标题文本。把 Date 列的 DataFormatString 设置为 {0:D}，只显示日期，不显示时间。

(2) 要编辑 GridView，必须用数据源定义一个 update 命令。选择名为 EventsDataSource 的 SqlDataSource 控件，再从智能标记中选择 Configure Data Source。在 Configure Data Source 对话框中，单击 Next 按钮，直到看到前面配置的 SELECT 命令为止。单击 Advanced 按钮，选中复选框 Generate INSERT, UPDATE, and DELETE statements，如图 18-34 所示。单击 OK 按钮，再单击 Next 和 Finish 按钮。

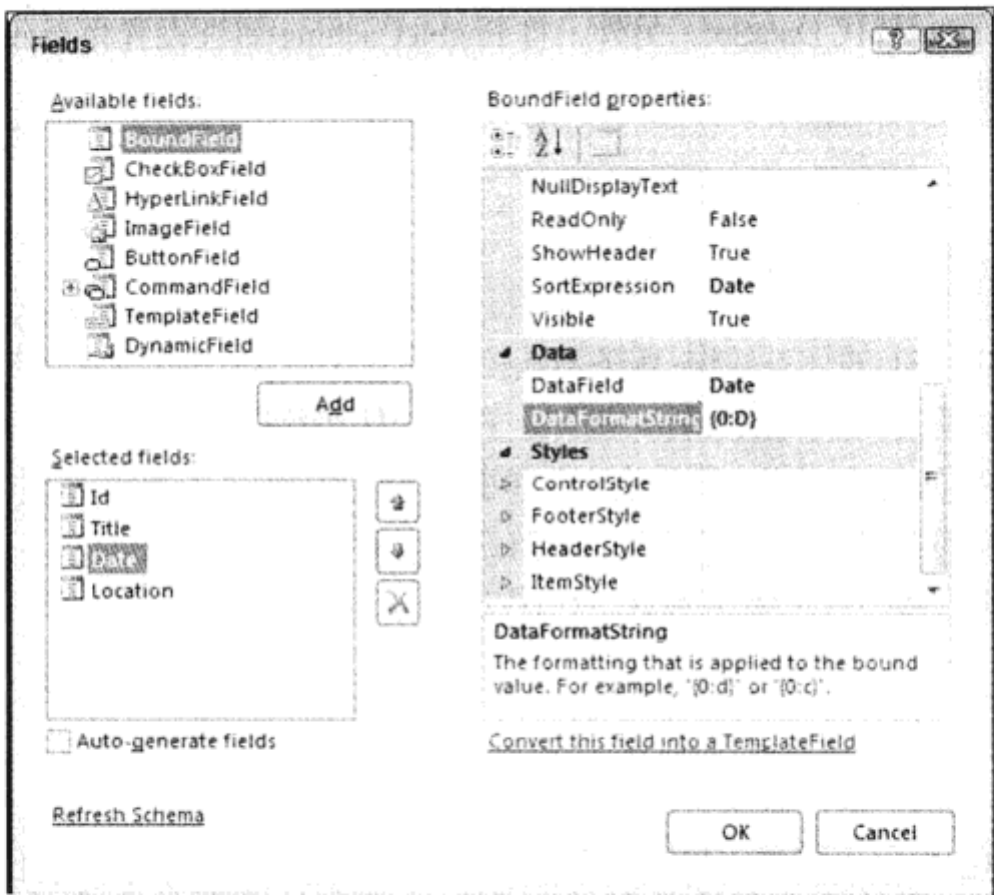


图 18-33

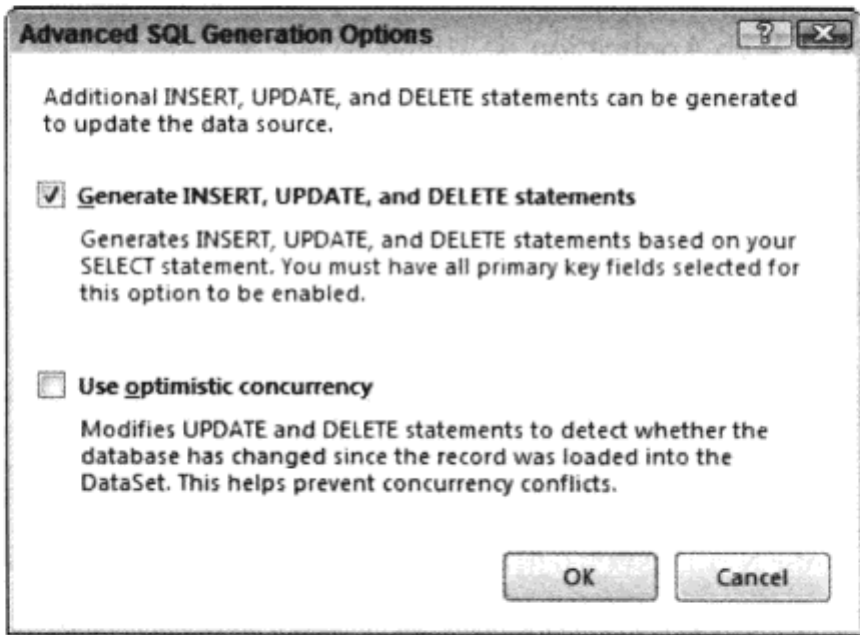


图 18-34

- (3) 再次选择 GridView 控件的智能标记。在智能标记菜单中有一个菜单项 Enable Editing。在选中复选框启动编辑后，GridView 控件中会添加一个新列。也可以用智能标记菜单编辑列，以便安排新 Edit 按钮的位置。另外选择 Enable Paging、Enable Sorting 和 Enable Selection 选项。
- (4) 启动应用程序，编辑已有的事件记录。单击一个标题，就会按该标题排序。

示例的说明

不必手工编写代码，使用 ASP.NET Web 控件就可以完成上述所有任务。这些控件在后台利用了许多功能。

例如，SqlDataSource 控件在 SqlDataAdapter 的帮助下，用数据库中的数据填充 DataSet。用于填充 DataSet 的数据是通过连接字符串和 SELECT 命令定义的。改变 SqlDataSource 的一个属性，就

可以使用 SqlDataReader 替代 DataSet。另外，将 EnableCaching 属性设置为 true，会自动使用 Cache 对象(本章前面讨论过)。

### 18.14 小结

本章学习了 ASP.NET 的体系结构、服务器端控件的用法和 ASP.NET 的一些基本功能。ASP.NET 提供了许多不需要编写太多代码的控件，例如，登录控件和数据控件。

学习了 ASP.NET 的基本功能、服务器控件和事件处理机制后，讨论了如何进行输入验证、状态管理的几种方法、身份验证和授权，以及显示数据库中的数据。

下面的练习帮助扩展本章开发的 Web 应用程序。

### 18.15 练习

(1) 在本章创建的母版页顶部添加用户名。可以使用 LoginName 控件。使用 LoginView，从而仅给通过身份验证的用户显示这个信息。

(2) 修改 Registration.aspx 页面的数据源，以使用 Events 数据库来显示事件。

(3) 创建一个 ASP.NET Web Application 类型的新项目。检查从这个项目模板中创建的所有文件和文件夹。它们看起来应很熟悉。

附录 A 给出了练习答案。

### 18.16 本章要点

主 题	重 要 概 念
使用 Web 服务器控件	Web 服务器控件是生成 HTML 代码的服务器端控件。这些控件的用法类似于 Windows 控件
使用 ASP.NET 回送	ASP.NET 回送模型在编写 ASP.NET Web 应用程序时是一个非常重要的概念。服务器端代码仅对服务器上的回送信息起作用。目前有了 ASP.NET Ajax，还可以定义 ASP.NET Ajax 回送，仅更新页面的一部分
通过验证控件验证用户的输入	ASP.NET 提供了几个验证控件，可以方便地验证客户端和服务端的用户输入。客户端的验证是为了提高性能，但因为 Web 客户从来都不是可信任的，所以还必须在服务器端进行验证
状态管理	在 Web 应用程序中，必须考虑在何处存储状态。状态可通过 cookie 或视图状态用于客户端，在服务器端，则通过会话、缓存和应用程序对象来使用状态
母版页	母版页用于包含多个页面中的公共部分
导航	菜单控件可用于在网站的不同页面之间导航。可以把站点地图绑定到菜单上，而无需把页面的链接直接添加到菜单控件上
读写 SQL Server 数据库	通过 ASP.NET 控件抽象出了数据库的访问。GridView 很容易在设计器上定制。这个网格的数据源只需设置属性，就可以在数据库中读写数据，而无需编写 C#代码



# 第 19 章

## Web 服务

本章内容:

---

- Web 服务概述
- 如何使用 ASP.NET 创建 Web 服务
- 如何通过 Windows 窗体应用程序使用 Web 服务
- 如何在 ASP.NET 客户程序中使用 Web 服务
- 如何异步调用 Web 服务
- 如何在 Web 服务之间传送数据

Web 应用程序是用户访问应用程序的功能的前端,而 Web 服务是应用程序访问应用程序的功能的前端。Web 服务是服务器端的程序,用以监听来自客户应用程序的消息,并返回特定的信息。这些信息可能来自 Web 服务本身,同一个域中的其他组件,或其他 Web 服务。

本章并不深入探讨 Web 服务的内部工作原理,但提供了足够多的通过 VS 开始创建和使用简单 ASP.NET Web 服务的信息。

### 19.1 使用 Web 服务的场合

为了从另一个角度来了解 Web 服务,需要区分用户—应用程序通信和应用程序—应用程序通信之间的区别。首先看看用户—应用程序通信,从 Web 上获得某些天气信息。[weather.msn.com](http://weather.msn.com) 和 [www.weather.com](http://www.weather.com) 等多个网站都提供了天气信息,其格式用户都很容易理解。用户可以直接阅读这些页面。

如果要创建一个富客户应用程序来显示天气(应用程序—应用程序通信),该应用程序必须连接到一个 Web 站点上,其 URL 字符串包含某个城市,我们想知道该城市的天气。接着要分析从 Web 站点返回的 HTML 信息,得到气温和天气信息,最后以合适的格式向富客户应用程序显示这些信息。

即使只想得到某个城市的气温信息,工作量也很大。而且,从 HTML 中获得数据的过程并不是很简单。这是因为 HTML 数据主要显示在 Web 浏览器上,并非供其他客户端商务应用程序使用。



因此，数据内嵌在文本中，不容易提取，需要重新编写或修改客户应用程序，从同一个 Web 页上提取不同的数据信息。而使用 Web 浏览器，用户可以立即得到需要的数据，忽略不需要的内容。

为了解决处理 HTML 数据的问题，Web 服务提供了一种有效的方式，可以只返回请求的数据。只要调用远程服务器上的一个方法，获得需要的信息，客户应用程序就可以直接使用这些信息了。我们不必处理对用户界面有意义的、预先格式化的文本，因为 Web 服务以 XML 格式显示信息，处理 XML 数据的工具早就有了。客户应用程序只需调用 .NET Framework XML 类的一些方法，就可以得到需要的数据。更妙的是，如果用 C# 编写 .NET Web 服务的客户应用程序，甚至不需要编写执行这些任务的代码，有一些可以自动生成 C# 代码的工具！

刚才讨论的天气应用程序是使用 Web 服务的一个示例，Web 服务还有其他许多用途。

### 19.1.1 宾馆旅行社代理应用程序

如何预订假日宾馆？不必求助于旅行社代理处，可以在 Internet 上预订假日宾馆。在某条航线的 Web 站点上，可以查找可能的航线，并预订它们。可以使用 Web 搜索引擎查找某个城市的宾馆。在许多情况下，还可以找到去宾馆的地图。在找到该宾馆的主页后，就可以导航到预订窗体页面，预订房间。接着可以搜索一个汽车租赁公司……

现在要做的是利用搜索引擎查找 Web 站点，接着浏览这些站点。另外，可以创建一个家庭旅行代理应用程序，它使用包含宾馆、航线和汽车租赁公司等信息的 Web 服务，给客户提供了一种方便的方式来处理所有的假日旅行问题，包括提前预订某场音乐会。在假日里使用移动设备，就可以使用相同的 Web 服务得到娱乐活动用的地图，文化活动或电影院的某些信息等。

### 19.1.2 图书发布应用程序

Web 服务还可以用于两个拥有合作关系的公司。假定图书出版商要得到书店销售图书的信息，就可以用一个 Web 服务来实现。可以建立一个使用该 Web 服务的 ASP.NET 应用程序，直接给用户提供服务。这个服务的另一个客户应用程序是书店的 Windows 应用程序，首先检查本地书店，再检查分销商的书店。销售员可以立即答出交货日期，无需用其他应用程序检查不同的书店。

### 19.1.3 客户应用程序的类型

Web 服务的客户应用程序可以是使用 Windows 窗体、WPF、Silverlight 创建的 Windows 应用程序，也可以是使用 Web 窗体创建的 ASP.NET 应用程序，Windows PC、UNIX 系统和移动设备都可以使用该 Web 服务。在 .NET Framework 中，Web 服务可以用于各类应用程序中。

## 19.2 应用程序的体系结构

使用 Web 服务的应用程序究竟是什么样的？无论是开发 ASP.NET 或 Windows 应用程序，还是开发小设备的应用程序，从前面介绍的应用可以看出，Web 服务的调用是非常类似的。在这些类型的应用程序中，Web 服务都是非常重要的技术。

图 19-1 说明了使用 Web 服务的方式。设备和浏览器通过 Internet 连接到一个用 Web 窗体开发的 ASP.NET 应用程序上。这个 ASP.NET 应用程序使用某些本地 Web 服务，以及可以通过网络获得的其他远程 Web 服务：门户 Web 服务、用于某个应用程序的 Web 服务和构件 Web 服务。下面的列

表详细描述了这些服务类型的含义：

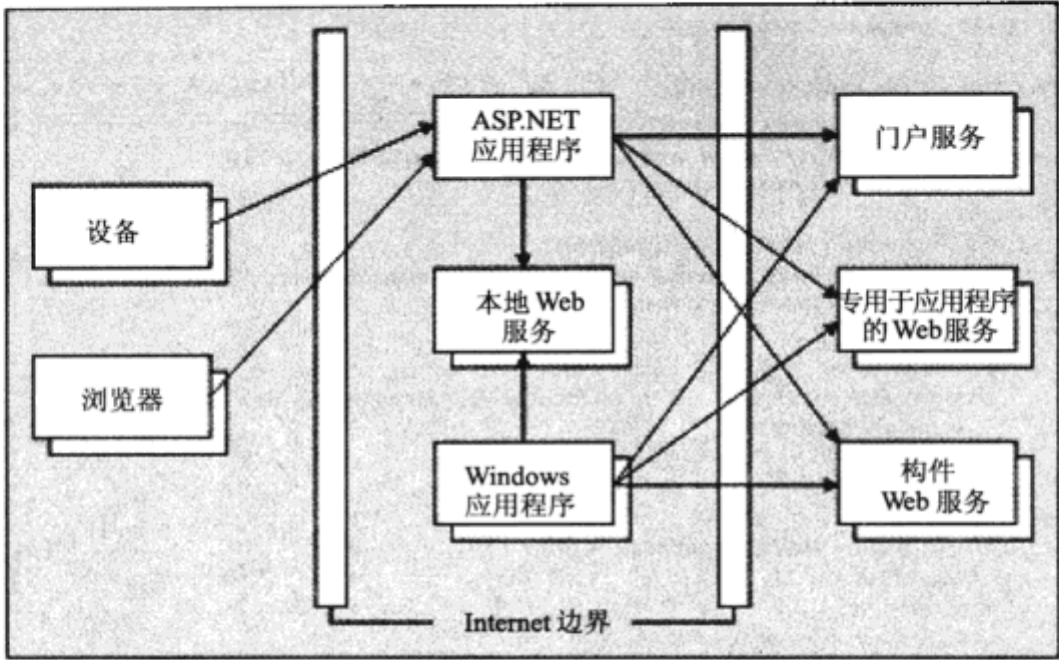


图 19-1

- 门户 Web 服务：使用同一个主题提供不同公司的服务。这将便于使用多个服务的入口点。
- 用于某个应用程序的 Web 服务：仅用于某个应用程序。
- 构件 Web 服务：可以在多个应用程序中使用。

图 19-1 中的 Windows 应用程序可以直接使用 Web 服务，无需通过 ASP.NET 应用程序。

### 19.3 Web 服务的体系结构

Web 服务利用 SOAP 协议，SOAP 协议是许多公司定义的标准。Web 服务的一个主要优点在于它的平台独立性。当需要在多个平台上协作时，Web 服务是一种有用的技术。在客户端和服务端开发 .NET 应用程序时，Web 服务也是一种很有用的技术。其优点是客户端和服务端都可以独立存在。服务描述是用 WSDL(Web 服务描述语言，Web Service Description Language)文档定义的，它可以用独立于 Web 服务新版本的方式设计，因此无需改变客户端。

下面详细介绍这一系列步骤。

#### 19.3.1 调用方法和 WSDL

WSDL 文档包含了下述信息：Web 服务支持什么方法，如何调用这些方法，给服务传送的参数类型，以及从服务返回的参数类型。图 19-2 显示了 ASP.NET 运行库生成的 WSDL。在 .asmx 文件的最后加上字符串?wsdl，以便返回一个 WSDL 文档。

不必直接处理这些信息。WSDL 文档是用 WebMethod 特性动态生成的；本章的后面会介绍这个属性。使用 Visual Studio 添加对客户应用程序的 Web 引用，就可以请求 WSDL 文档。这个 WSDL 文档用于创建带有相同方法和参数的客户代理程序。而利用这个代理程序，客户应用程序就只需在服务器中执行时调用方法，因为代理程序会把它转换为 SOAP 调用，通过网络进行调用。

WSDL 规范由 World Wide Web Consortium(W3C)维护，可以在 W3C 网站 [www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl) 上阅读这个规范。

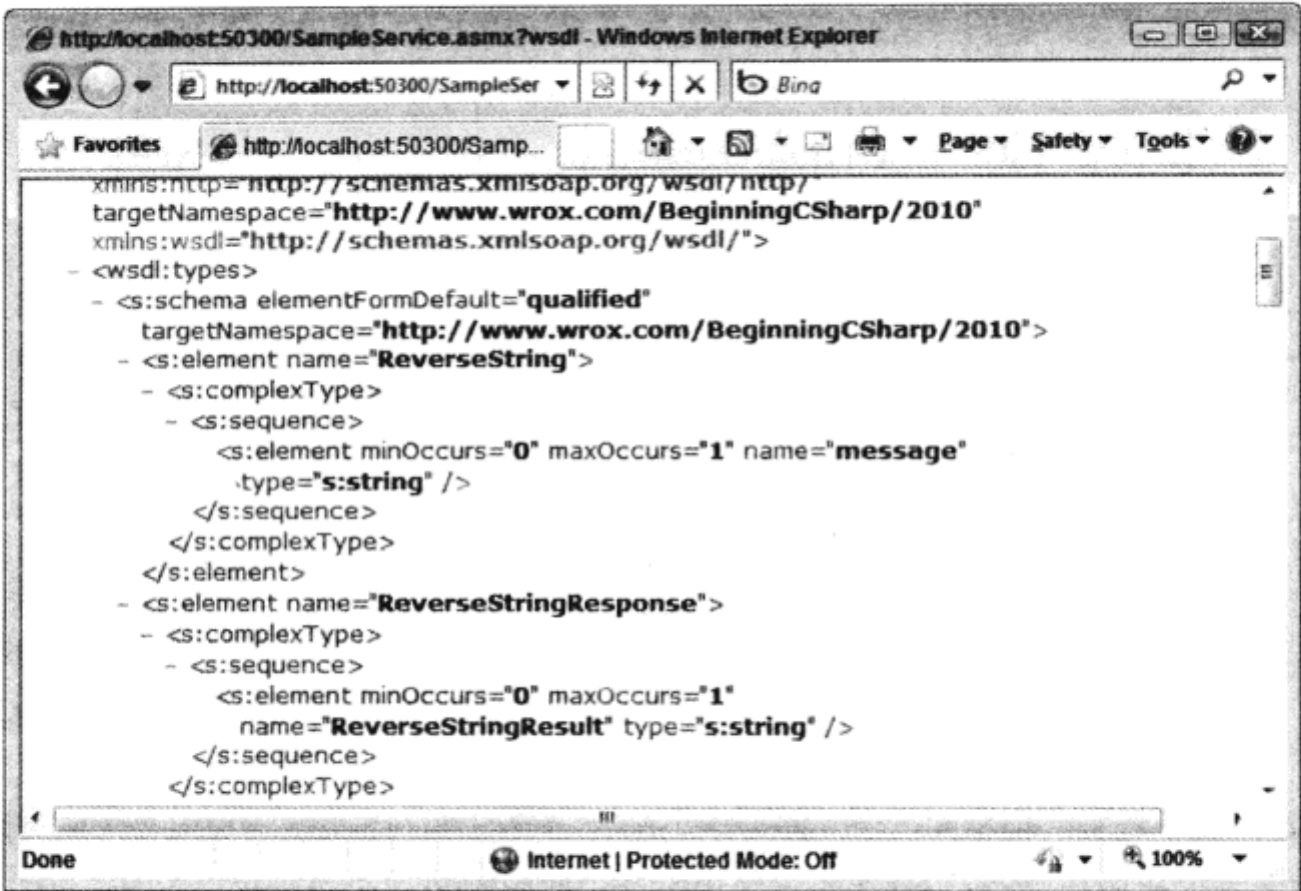


图 19-2

19.3.2 调用方法

SOAP 消息是客户机和服务器之间的基本通信单元。要调用 Web 服务上的一个方法，该调用必须转换为 SOAP 消息，因为它是在 WSDL 文档中定义的。图 19-3 显示了 SOAP 消息的一部分。SOAP 封套(envelop)把所有的 SOAP 消息封装在一个块中。SOAP 封套本身由两部分组成：SOAP 标题和 SOAP 体。标题是可选的，它定义了客户机和服务器应如何处理 SOAP 体。SOAP 体是必须有的，它包括发送的数据，通常 SOAP 体中的信息是要调用的方法和序列化的参数值。SOAP 服务器在 SOAP 消息的消息体中返回值。



图 19-3

在下面的示例中，说明了从客户机发送给服务器的 SOAP 消息是什么样的。客户机调用 Web 服务方法 ReverseString()。字符串 Hello World! 作为这个方法的参数传送。该方法调用位于 SOAP 体中，位于 XML 元素<soap:Body>中。消息体本身包含在封套<soap:Envelope>中。在 SOAP 消息的开头，有 HTTP 标题，因为 SOAP 消息通过 HTTP POST 请求来发送。  
未必要创建这样一个消息，因为它可以由客户代理程序创建：

```

POST /Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 508
SOAPAction: "http://www.wrox.com/webservices/ReverseString"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseString xmlns="http://www.wrox.com/webservices">
      <message>Hello World!</message>
    </ReverseString>
  </soap:Body>
</soap:Envelope>

```

服务器用 SOAP 消息!dlroW olleH 作为响应, 如 XML 元素 ReverseStringResult 所示:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 446

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseStringResponse xmlns="http://www.wrox.com/webservices">
      <ReverseStringResult>!dlroW olleH</ReverseStringResult>
    </ReverseStringResponse>
  </soap:Body>
</soap:Envelope>

```

SOAP 规范由 W3C 的 XML Protocol Working Group 维护(请访问 [www.w3.org/TR/soap](http://www.w3.org/TR/soap))。

### 19.3.3 WS-I 规范


SOAP 规范和其他基于 SOAP 的规范已经存在一段时间了。其间有很多变化, 出现了多个版本, 从而增大了交互难度。为了解决这个问题, 成立了 Web 服务交互操作组织(Web Services Interoperability Organization (<http://ws-i.org>))。这个组织用 WS-I Basic Profile 规范定义了 Web 服务的需求。WS-I Basic Profile 详见 [www.ws-i.org](http://www.ws-i.org)。用 ASP.NET 开发的 Web 服务遵循 Basic Profile 1.1, 它在 <http://www.ws-i.org/Profiles/BasicProfile-1.1.html> 文档中定义。

## 19.4 Web 服务和 .NET Framework

在 .NET Framework 中, 很容易创建和使用 Web 服务。下面列出的 3 个主要名称空间可以处理 Web 服务:

- System.Web.Services 名称空间中的类用于创建 Web 服务。
- 使用名称空间 System.Web.Services.Description, 可以通过 WSDL 描述 Web 服务。

- 使用 System.Web.Services.Protocols，可以创建 SOAP 请求和响应。



可以使用 ASP.NET 或 WCF 创建 Web 服务。WCF 比 ASP.NET Web 服务灵活得多，因为它给 Web 服务的驻留提供了不同选项，而不仅限于 ASP.NET，还支持不同的协议，而不仅限于 HTTP。ASP.NET Web 服务的优点是更容易使用。另外，WCF 模板不能在 VS2008 的 Express 版本中使用，在编写本书时，WCF 模板能否在 VS2010 中使用还不清楚。第 27 章介绍 WCF。

19.4.1 创建 Web 服务

要实现 Web 服务，可以从 System.Web.Services.WebService 中派生 Web 服务类。WebService 类提供了对 ASP.NET Application 和 Session 对象的访问，这个类是可选的，只有需要访问该类提供的属性时，才需要从这个类中派生。

表 19-1 列出了 WebService 类的一些属性。

表 19-1

属 性	说 明
Application	为当前请求返回一个 HttpApplicationState 对象
Context	返回一个封装 HTTP 特定信息的 HttpContext 对象。在这里，可以读取 HTTP 标题信息
Server	返回一个 HttpServerUtility 对象。这个类有一些帮助方法，可以进行 URL 编码和解码
Session	返回一个 HttpSessionState 对象，以存储客户机的一些状态信息
User	返回一个实现接口 IPrincipal 的用户对象。使用这个接口可以得到用户名和身份验证类型
SoapVersion	返回 Web 服务使用的 SOAP 版本。SOAP 版本封装在 SoapProtocolVersion 枚举中

1. WebService 特性

用特性 WebService 来标记 WebService 的子类。WebServiceAttribute 类有下述属性，如表 19-2 所示。

表 19-2

属 性	说 明
Description	服务的描述信息，可用于 WSDL 文档
Name	获取或设置 Web 服务名称
Namespace	获取或设置 Web 服务的 XML 名称空间。其默认值是 http://tempuri.org，它用于测试，但在公开这个服务前，应修改该名称空间

2. WebMethod 特性

Web 服务中可供使用的所有方法都必须用 WebMethod 特性来标记。当然，服务还可以有其他



没有用 WebMethod 标记的方法。这些方法可以从 WebMethod 中调用，但不能在客户机上调用。使用特性类 WebMethodAttribute，就可以在远程客户机上调用方法，并可以定义是否缓存响应，缓存的有效使用时间，会话状态是否与指定的参数一起存储。表 19-3 列出了 WebMethod Attribute 类的属性。

表 19-3

属 性	说 明
BufferResponse	获取或设置表示是否应缓存响应的标志。默认值为 true。使用被缓存的响应，仅可以将已完成的数据包传递给客户机
CacheDuration	设置结果应缓存的时长。如果在这个属性设置的时间段中第二次发出了相同的请求，就返回缓存的值。默认值为 0，这表示不缓存结果
Description	用于给预期的用户生成服务帮助页面
EnableSession	布尔值，表示会话状态是否有效。默认值是 false，因此 WebService 类的 Session 属性不能用于存储会话状态
MessageName	默认状态下，把消息名设置为方法名
TransactionOption	这个属性表示方法的事务处理支持。默认值是 Disabled

3. WebServiceBinding 特性

特性 WebServiceBinding 用于把 Web 服务标记为可交互操作的一致性级别。表 19-4 中列出了 WebServiceBindingAttribute 类的一些属性。

表 19-4

属 性	说 明
ConformsTo	设置为 WsiProfile 枚举的一个值。WsiProfile 有两个值：Web 服务遵循 Basic Profile 1.1 时，其值为 BasicProfile1_1；没有定义任何一致性级别时，其值为 None
EmitConformanceClaims	一个布尔属性，定义了用 ConformanceClaims 属性指定的一致性级别是否应传送给生成的 WSDL 文档
Name	定义绑定的名称。该名称默认情况下与 Web 服务相同，但末尾加上了 Soap 字符串
Location	定义了绑定消息的位置，例如 http://www.wrox.com/DemoWebservice.asmx?wsdl
Namespace	定义了绑定的 XML 名称空间

19.4.2 客户程序

要调用一个方法，客户机必须创建一个与 Web 服务所在的服务器之间的 HTTP 连接，并发送一个 HTTP 请求，以便传送 SOAP 消息。方法调用必须转换为 SOAP 消息。这些都是由客户代理程序实现的。客户代理程序的实现代码在 SoapHttpClientProtocol 类中。

1. SoapHttpClientProtocol

类 System.Web.Services.Protocols.SoapHttpClientProtocol 是客户代理程序的基类。Invoke()方法转换参数，建立一个 SOAP 消息，发送给 Web 服务。调用哪个 Web 服务由 Url 属性确定。

SoapHttpClientProtocol 类也支持使用 BeginInvoke()和 EndInvoke()方法的异步调用。

2. 其他客户协议

如果不使用 SoapHttpClientProtocol 类，还可以使用其他代理程序类，HttpGetClientProtocol 和 HttpPostClientProtocol 仅执行简单的 HTTP GET 或 HTTP POST 请求，没有 SOAP 调用的系统开销。

如果解决方案在客户机和服务器上使用 .NET，就可以使用 HttpGetClientProtocol 和 HttpPostClientProtocol 类。如果要使用不同的技术，就必须使用 SOAP 协议。

比较 HTTP POST 请求和本章前面介绍的 SOAP 调用：

```
POST /WebServiceSample/Service1.asmx/ReverseString HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

message=string
```

HTTP GET 请求甚至更短，GET 请求的缺点是发送的参数长度是有限制的。如果其长度超过了 1K，就应考虑使用 POST：

```
GET /WebServiceSample/Service1.asmx/ReverseString?message=string HTTP/1.1
Host: localhost
```

HttpGetClientProtocol 和 HttpPostClientProtocol 的开销比 SOAP 方法小，但缺点是其他平台上的 Web 服务不支持它，且只能发送简单数据。

19.5 创建简单的 ASP.NET Web 服务

下面用 Visual Studio 创建一个简单的 Web 服务。

试一试：创建一个 Web 服务项目

(1) 选择 File | New | Project，再选择 ASP.NET Empty Web Application 模板，创建一个新的 Web 服务项目，如图 19-4 所示。把项目命名为 WebServiceSample，单击 OK 按钮。

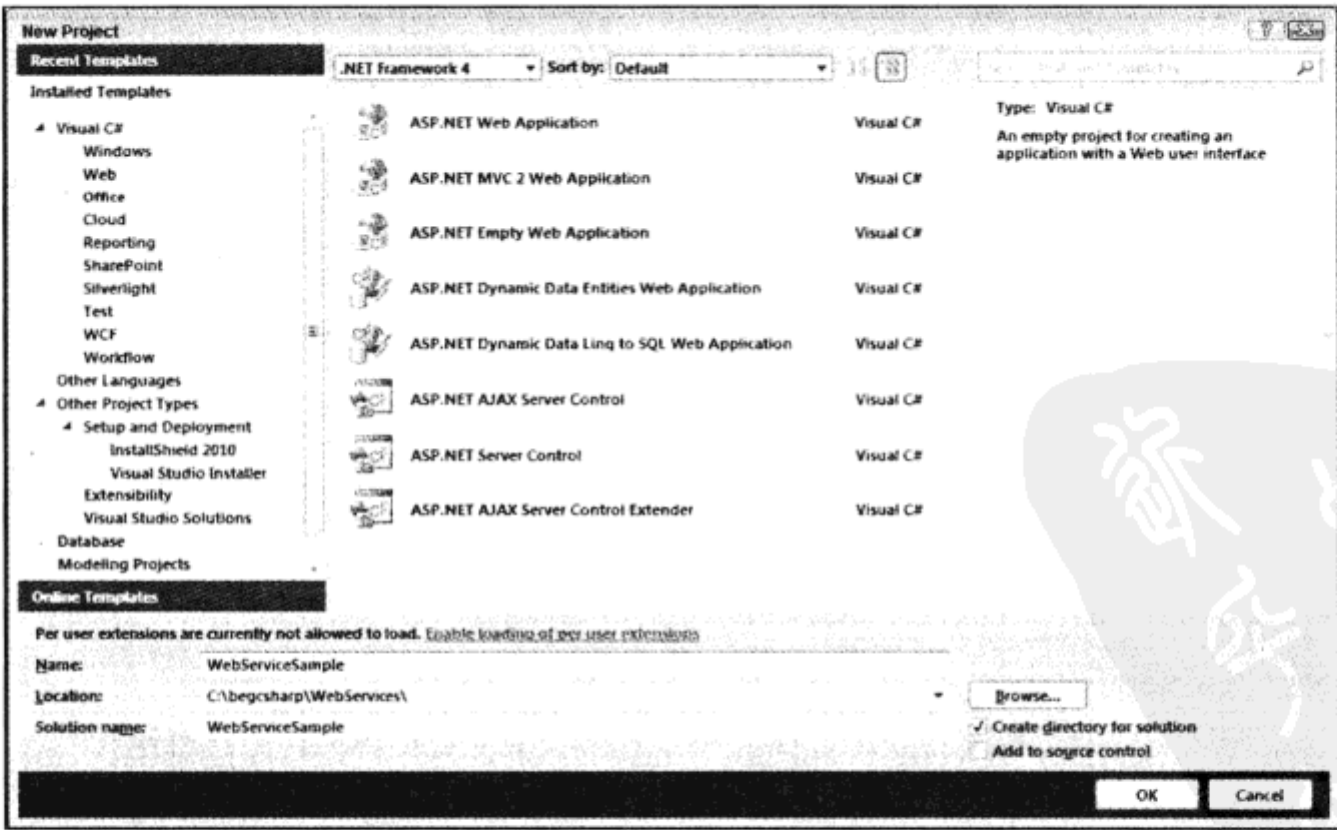


图 19-4



(2) 添加一个新项，选择 Web Service 模板，把所创建的文件命名为 SampleService.asmx，如图 19-5 所示。

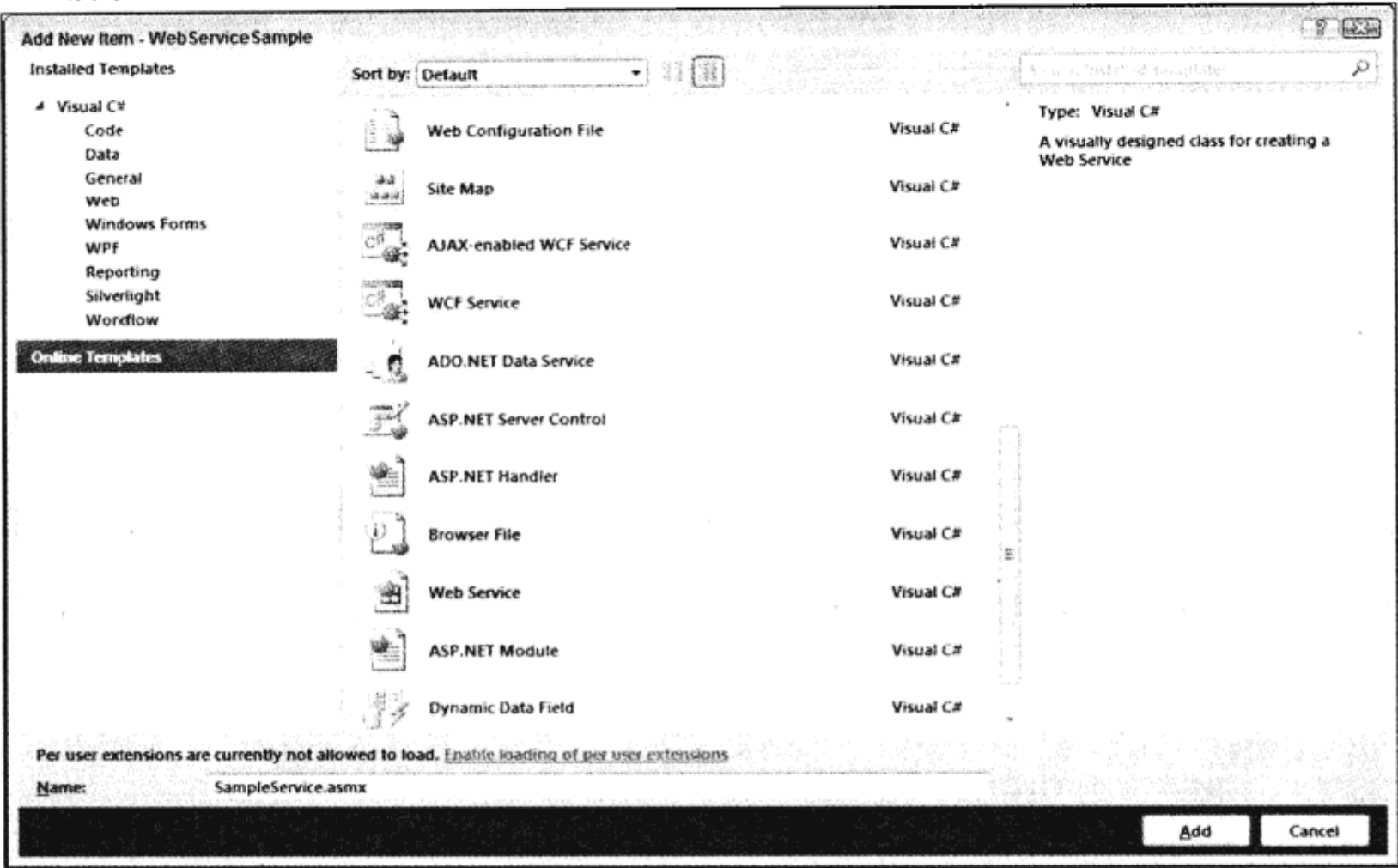


图 19-5

示例的说明

项目模板和项模板生成的文件如下所示：

- **SampleService.asmx**：保存 Web 服务类。所有的 ASP.NET Web 服务都用.asmx 扩展名来标识。包含源代码的文件是 SampleService.asmx.cs，因为在 Visual Studio 中可以使用后台编码功能。
- **SampleService.asmx.cs**：项模板在该文件中生成了一个派生自 System.Web.Services.WebService 的类 SampleService。在这个文件中，一些示例代码说明了 Web 服务的方法是如何编码的——它应是公共的，用 WebMethod 特性标记：



可从  
wtox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

namespace WebServiceSample
{
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
    public class SampleService : System.Web.Services.WebService
    {
        [WebMethod]
```

```
        public string HelloWorld()
        {
            return "Hello World";
        }
    }
}
```

代码段 WebServiceSample/SampleService.asmx.cs

添加 Web 方法

接着给 Web 服务添加一个定制方法。这里添加一个简单方法 ReverseString(), 它接收一个字符串, 颠倒该字符串中的字符顺序后, 将其返回给客户机。

试一试: 添加一个方法

(1) 删除方法 HelloWorld()及其全部实现代码。在 SampleService.asmx.cs 文件中添加下述代码。



可从  
Wrox.com  
下载源代码

```
[WebMethod]
public string ReverseString(string message)
{
    return new string(message.Reverse().ToArray());
}
```

代码段 WebServiceSample/SampleService.asmx.cs

(2) 修改 SampleService.asmx.cs 文件中的示例代码, 如下所示:

```
[WebService(Namespace = "http://www.wrox.com/BeginningCSharp/2010")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class SampleService : System.Web.Services.WebService
```

代码段 WebServiceSample/SampleService.asmx.cs

(3) 编译项目。

示例的说明

ASP.NET 运行库使用反射技术读取 Web 服务专用的一些特性, 例如[WebMethod], 把方法提供为 Web 服务的操作。ASP.NET 运行库还提供了 WSDL 来描述服务。

要在为 Web 服务生成的描述信息中唯一地标识 XML 元素, 应添加一个 XML 名称空间。这个示例在类 Service 中用[WebService]特性添加了名称空间 http://www.wrox.com/webservices。当然, 可以使用其他字符串唯一地标识 XML 元素, 例如, 公司页面的 URL 链接。Web 链接不一定存在, 它仅用于唯一标识。如果使用基于公司网址的名称空间, 就可以确保其他公司不会使用这个名称空间。

如果没有改变 XML 名称空间, 使用的默认名称空间是 http://tempuri.org。从学习的角度来看, 这个默认的名称空间就已足够了, 但不应部署一个使用它的 Web 服务产品。

## 19.6 测试 Web 服务

现在就可以测试服务了。在浏览器中打开文件 `Service1.asmx`(可以选择 `Debug | Start Without Debugging`, 在 Visual Studio 2010 中启动它), 列出服务中的所有方法, 如图 19-6 所示。在该服务中, 只有一个方法 `ReverseString()`。

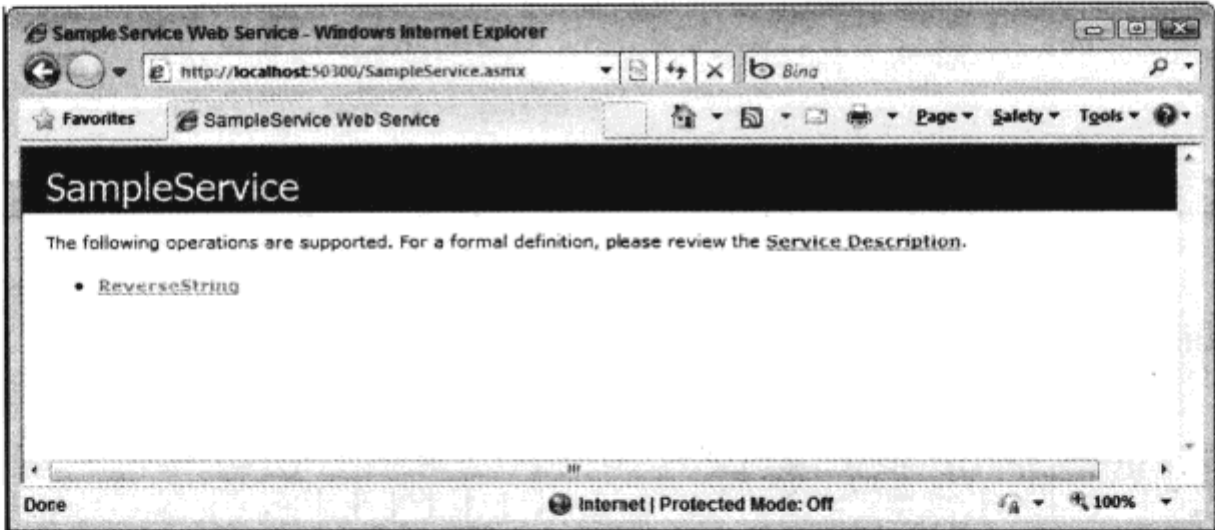


图 19-6

选择 `ReverseString` 方法的链接, 显示一个对话框, 以测试 Web 服务。该测试对话框已经编辑了传递给 `ReverseString` 方法的参数的字段, 在本例中, 该方法只有一个参数。

在这个页面中, 还可以获得客户机中的 SOAP 调用信息, 以及服务器返回的响应信息, 如图 19-7 所示。本示例显示 SOAP 和 HTTP POST 请求。

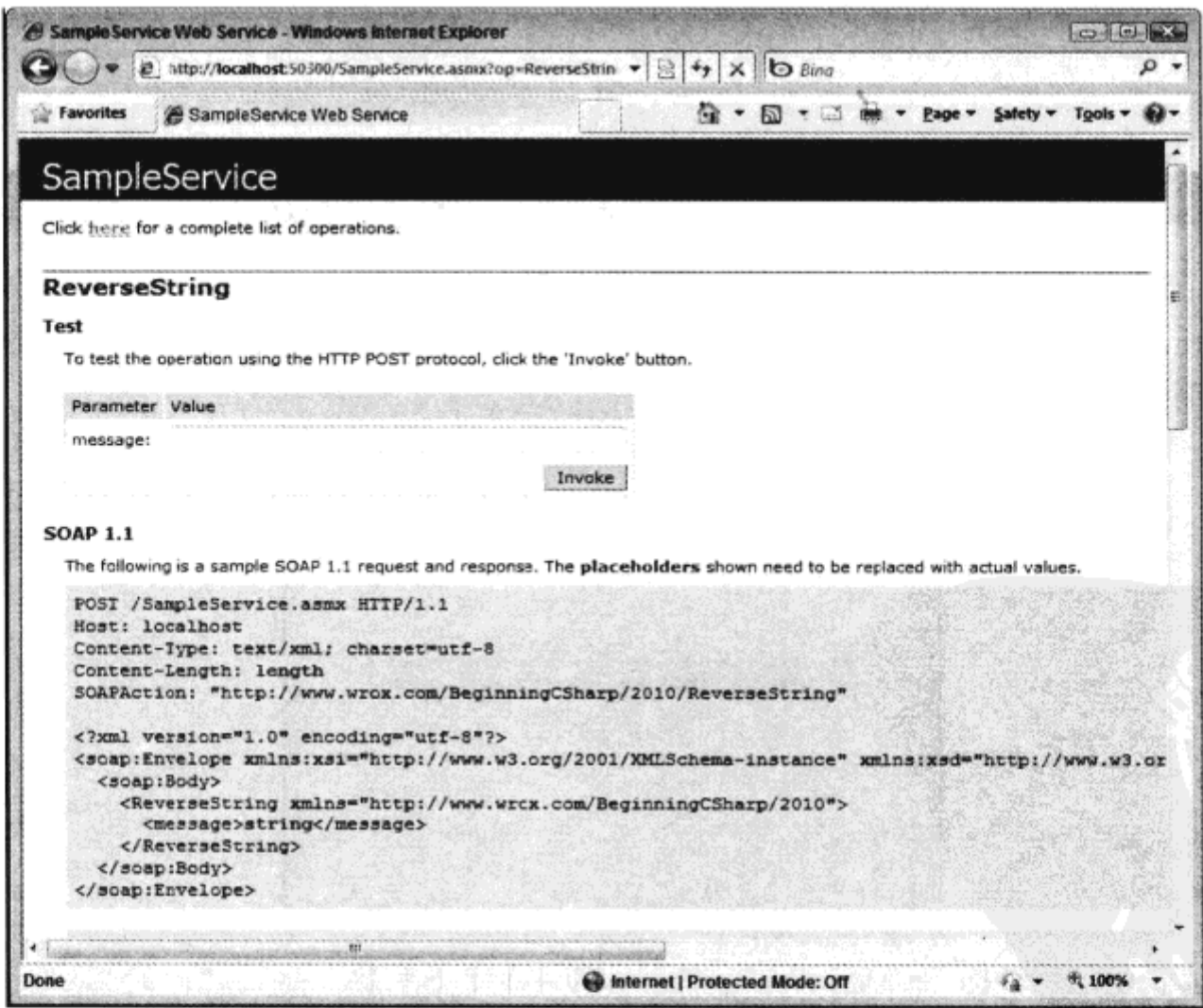


图 19-7

在把字符串 Hello Web Services! 写到文本框中后，单击 Invoke 按钮，服务器就会发送如图 19-8 的结果。

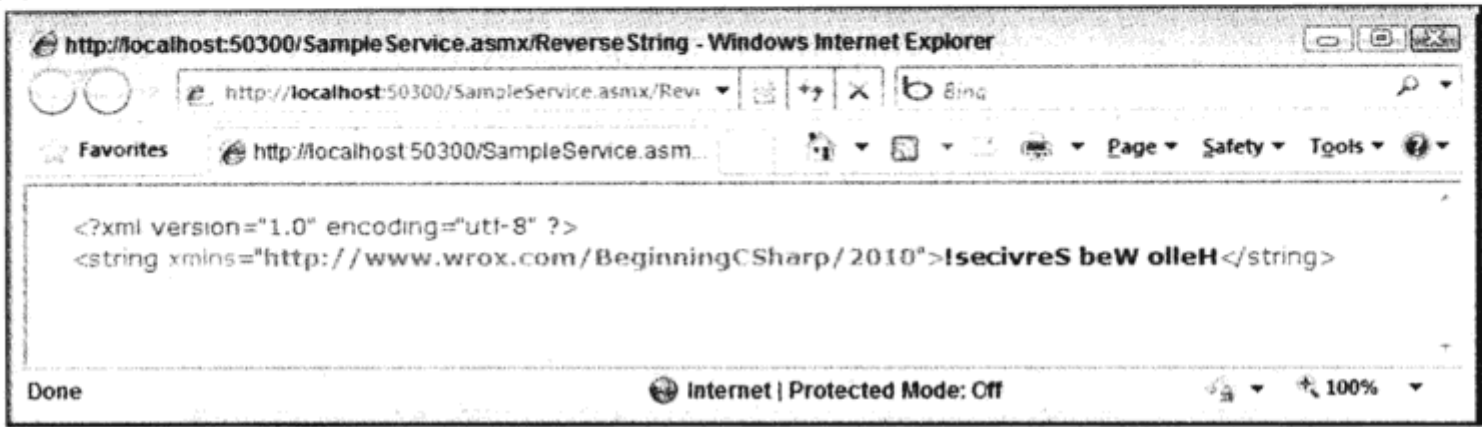


图 19-8

结果是 string 类型，与预期的一样，它的顺序与我们输入的字符串的顺序相反。

### 19.7 实现 Windows 客户程序

测试工作进展顺利，但我们希望创建一个使用 Web 服务的 Windows 客户程序。该客户程序必须创建一个 SOAP 消息，通过 HTTP 通道发送出去。这不一定要我们自己去完成。VS2010 创建了一个代理类，它使用 WCF 中的 HTTP 信道，在后台完成了所有的工作。

有关 WCF 的具体内容请参见第 26 章。

#### 试一试：创建一个 Windows 客户程序

(1) 在已有的解决方案 WebServiceSample 中添加一个新的 C# Windows 窗体应用程序 WindowsFormsClient。在窗体上添加两个文本框和一个按钮，如图 19-9 所示。您将利用按钮的单击事件处理程序调用该 Web 服务。



图 19-9

(2) 要添加一个服务引用，选择 Project | Add Service Reference 菜单项。在该对话框中，单

击 Discover 按钮箭头，选择 Services in Solution。前面创建的 Web 服务就会显示在左边的视图中。在左边的树形视图中选择 SampleServiceSoap。在单击 OK 按钮之前，将 Namespace 名称改成 WebServicesSample，如图 19-10 所示。

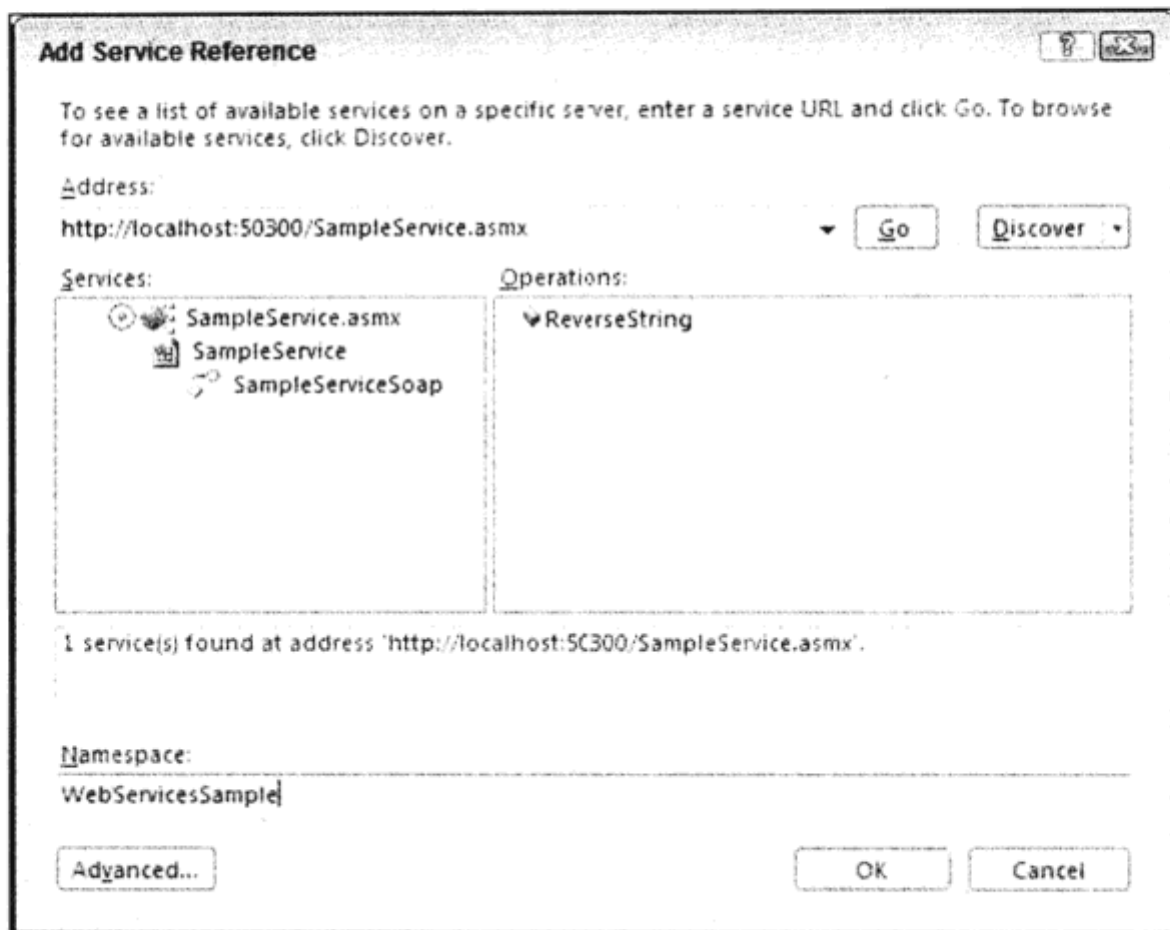


图 19-10

(3) 前面一直都没有为客户程序编写代码，而是设计了一个小用户界面，使用 Add Service Reference 菜单创建了一个代理类。现在要创建两者之间的链接。给按钮添加一个 Click 事件处理程序 button1\_Click()，并添加如下两条语句：



可从  
wrox.com  
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    var client = new WebServicesSample.SampleServiceSoapClient();
    textBox2.Text = client.ReverseString(textBox1.Text);
}
```

代码段 WindowsFormsClient/Form1.cs

### 示例的说明

在 Solution Explorer 中，可以看到一个新服务引用 WebServicesSample，如图 19-11 所示。单击 Show All Files 按钮后，可以看到 WSDL 文档和文件 Reference.cs，该文件包含代理程序的源代码。Show All Files 按钮是 Solution Explorer 工具栏中的第二个按钮。把鼠标指针移动到这些按钮上，就会显示工具提示，给出每个按钮的相关信息。

单击 Show All Files 按钮后在 Solution Explorer 中显示的内容，使用 Class View 更容易看到(实现客户代理程序的新类)。该类把方法调用转换为 SOAP 格式。在 Class View(如图 19-12 所示)中，有一个使用 Web Reference 名称定义的新名称空间。本例创建了 WebServicesSample。类 SampleServiceSoapClient 派生自 ClientBase<SampleServiceSoap>，实现 Web 服务的 ReverseString()

方法。

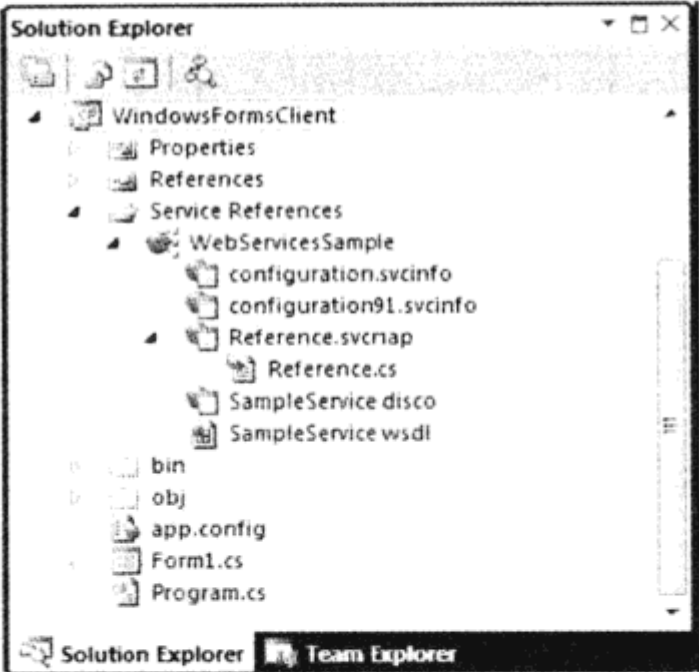


图 19-11

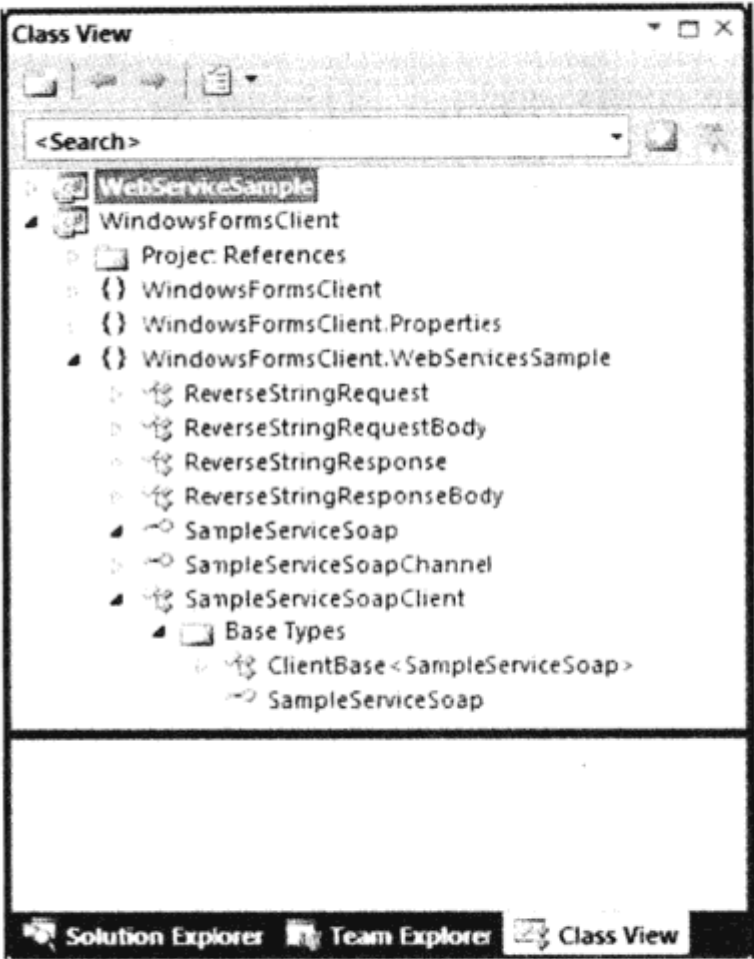


图 19-12

双击 `SampleServiceSoapClient` 类，打开自动生成的 `Reference.cs` 文件，下面看看生成的代码。  
类 `SampleServiceSoapClient` 派生自 `ClientBase<SampleServiceSoap>`。这个基类在 `Invoke()` 方法中创建了一个 SOAP 消息。`SampleServiceSoap` 是一个接口，定义了 Web 服务的所有操作。



可从  
wrox.com  
下载源代码

```
[System.Diagnostics.DebuggerStepThroughAttribute()]  
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",  
    "4.0.0.0")]  
public partial class SampleServiceSoapClient : ClientBase<SampleServiceSoap>,  
    SampleServiceSoap {
```

代码段 WindowsFormsClient/Service References/WebServicesSample/Reference.svcmap/Reference.cs

最重要的方法是 Web 服务提供的：`ReverseString()`。该方法与服务器上实现的方法有相同的参数。`ReverseString()` 客户端版本的实现代码调用基类 `SoapHttpClientProtocol` 的 `Invoke()` 方法。`Invoke()` 使用方法名 `ReverseString` 和参数 `message` 创建一个 SOAP 消息。这个方法在文件 `reference.cs` 中：



可从  
wrox.com  
下载源代码

```
public string ReverseString(string message) {  
    ReverseStringRequest inValue = new ReverseStringRequest();  
    inValue.Body = new ReverseStringRequestBody();  
    inValue.Body.message = message;  
    ReverseStringResponse retVal =  
        ((SampleServiceSoap) (this)).ReverseString(inValue);  
    return retVal.Body.ReverseStringResult;  
}
```

代码段 WindowsFormsClient/Service References/WebServicesSample/Reference.svcmap/Reference.cs



通过 HTTP 发送的 SOAP 请求是在自动创建的应用程序配置文件中定义的，它定义了 `basicHttpBinding`：



可从  
wrox.com  
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="SampleServiceSoap" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
          maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
          useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None" proxyCredentialType="None"
              realm="" />
            <message clientCredentialType="UserName" algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:50300/SampleService.asmx"
        binding="basicHttpBinding" bindingConfiguration="SampleServiceSoap"
        contract="WebServicesSample.SampleServiceSoap" name="SampleServiceSoap" />
    </client>
  </system.serviceModel>
</configuration>
```

代码段 WindowsFormsClient/app.config

对服务的调用是在下面的语句中使用生成的代理类实现的。利用下面这行代码，创建代理类的一个新实例。在构造函数的实现代码中，`Url` 属性设置为 Web 服务：



可从  
wrox.com  
下载源代码

```
var client = new WebServicesSample.SampleServiceSoapClient();
```

代码段 WindowsFormsClient/Form1.cs

调用代理类的 `ReverseString()` 方法，把 SOAP 消息发送给服务器，以调用 Web 服务：



可从  
wrox.com  
下载源代码

```
textBox2.Text = client.ReverseString(textBox1.Text);
```

代码段 WindowsFormsClient/Form1.cs

运行程序，会得到如图 19-13 所示的输出结果。





图 19-13

### 19.8 异步调用服务

在通过网络发送消息时，总是要注意网络的延迟时间。如果 Web 服务是同步调用的，客户应用程序就会在调用返回后再继续执行。这在本地网络上速度比较快，但必须注意生产系统的网络体系架构。

可以给 Web 服务异步发送消息。客户代理程序不仅会创建同步方法，还可以创建异步方法。但是，Windows 应用程序存在一个特殊的问题，即每个 Windows 控件都会被绑定到一个线程上，而 Windows 控件的方法和属性只能在创建它的线程中调用。NET 4 的代理类对此问题有特定的解决方法，如下面生成的代理代码所示。

试一试：异步调用服务

要使用代理类的异步实现代码，请执行以下步骤：

- (1) 修改已生成的代理类：选择服务引用 WebServicesSample。打开关联菜单，选择 Configure Service Reference，打开的对话框如图 19-14 所示。

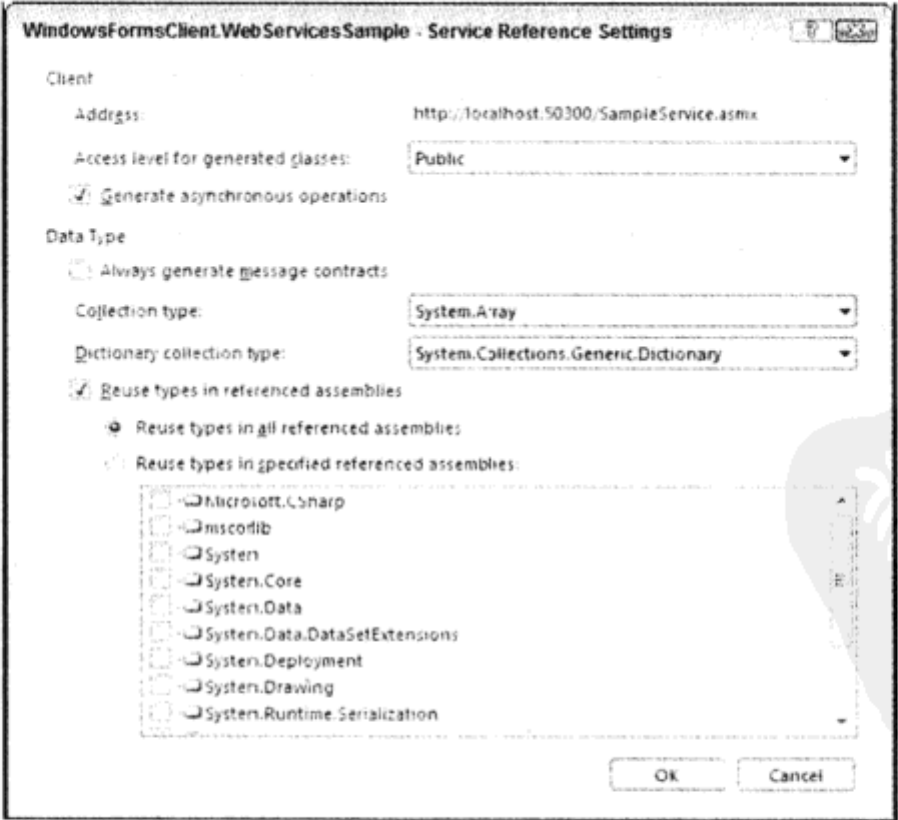


图 19-14

(2) 在 Service Reference Settings 对话框中选中 Generate asynchronous operations 复选框。

(3) 要异步调用 Web 服务, 应修改 button1\_Click 方法的实现代码(如下所示)。在初始化代理后, 给事件 ReverseStringCompleted 添加事件处理程序 client\_ReverseStringCompleted。接着调用代理 ReverseStringAsync 的 async 方法, 传送 textBox1 的 Text 属性。在 async 方法中, 会创建一个线程, 以调用 Web 服务。



可从  
wrox.com  
下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    var client = new WebServicesSample.SampleServiceSoapClient();
    client.ReverseStringCompleted += client_ReverseStringCompleted;
    client.ReverseStringAsync(textBox1.Text);
}
```

代码段 WindowsFormsClient/Form1.cs

(4) 现在实现处理方法 client\_ReverseStringCompleted:



可从  
wrox.com  
下载源代码

```
void client_ReverseStringCompleted(object sender,
                                   ReverseStringCompletedEventArgs e)
{
    if (e.Error != null)
    {
        textBox2.Text = e.Result;
    }
    else
    {
        MessageBox.Show(e.Error.Message);
    }
}
```

代码段 WindowsFormsClient/Form1.cs

会在 Web 服务调用完成后再调用这个方法。在实现代码中, ReverseStringCompletedEventArgs 参数的 Result 属性被传送给 textBox2 的 Text 属性。

(5) 现在就可以再次运行客户程序, 测试异步调用了。还可以在 Web 服务的实现中添加一段休眠时间, 这样就可以看到, 客户应用程序的 UI 在调用 Web 服务的过程中没有停止。

#### 示例的说明

下面的代码是 ReverseString() 方法的异步版本。在代理类的异步实现代码中, 总是有一个方法可以异步调用, 也总是有一个事件可以确定在完成 Web 服务方法时调用哪个方法。

ReverseStringAsync() 方法只有发送给服务器的参数。从客户端接收的数据可以通过把事件处理程序赋予 ReverseStringCompleted 事件来读取。这个事件的类型是 EventHandler <ReverseStringCompletedEventArgs>。该类型是一个委托, 其中第二个参数 ReverseStringCompletedEventArgs 是从 ReverseString() 方法的输出参数中创建的。类 ReverseStringCompletedEventArgs 在 Result 属性中包含 Web 服务的返回数据。这段执行代码能正常工作是因为使用了 SendOrPostCallback 委托, 这个委托把调用转发给 Windows Forms 控件的正确线程。



可从  
wrox.com  
下载源代码

```

public event System.EventHandler<ReverseStringCompletedEventArgs>
    ReverseStringCompleted;

public void ReverseStringAsync(string message) {
    this.ReverseStringAsync(message, null);
}

public void ReverseStringAsync(string message, object userState) {
    if ((this.onBeginReverseStringDelegate == null)) {
        this.onBeginReverseStringDelegate =
            new BeginOperationDelegate(this.OnBeginReverseString);
    }
    if ((this.onEndReverseStringDelegate == null)) {
        this.onEndReverseStringDelegate =
            new EndOperationDelegate(this.OnEndReverseString);
    }
    if ((this.onReverseStringCompletedDelegate == null)) {
        this.onReverseStringCompletedDelegate =
            new SendOrPostCallback(this.OnReverseStringCompleted);
    }
    base.InvokeAsync(this.onBeginReverseStringDelegate, new object[] {
        message}, this.onEndReverseStringDelegate,
        this.onReverseStringCompletedDelegate, userState);
}

private void OnReverseStringCompleted(object state) {
    if ((this.ReverseStringCompleted != null)) {
        InvokeAsyncCompletedEventArgs e =
            ((InvokeAsyncCompletedEventArgs)(state));
        this.ReverseStringCompleted(this,
            new ReverseStringCompletedEventArgs(e.Results, e.Error,
            e.Cancelled, e.UserState));
    }
}

public partial class ReverseStringCompletedEventArgs :
    AsyncCompletedEventArgs {

    private object[] results;

    public ReverseStringCompletedEventArgs(object[] results,
        Exception exception, bool cancelled, object userState) :
        base(exception, cancelled, userState) {
        this.results = results;
    }

    public string Result {
        get {
            base.RaiseExceptionIfNecessary();
            return ((string)(this.results[0]));
        }
    }
}

```

代码段 WindowsFormsClient/Service References/WebServicesSample/Reference.svcmap/Reference.cs

## 19.9 实现 ASP.NET 客户程序

现在可以在 ASP.NET 客户应用程序中使用相同的服务。引用 Web 服务的方式与处理 Windows Forms 应用程序的方式相同。

试一试：创建 ASP.NET 客户应用程序

- (1) 打开前面创建的 Web 服务 WebServicesSample。
- (2) 在解决方案中添加一个新 C# ASP.NET Empty Web Application，命名为 ASPNETClient。
- (3) 添加一个新的 Web 窗体 Default.aspx，在 Web 窗体中添加两个文本框和一个按钮，如图 19-15 所示。



图 19-15

- (4) 采用前面处理 Windows 应用程序的方式，添加一个对 `http://localhost:50300/sample.service.asmx` 的服务引用。根据自己的配置，可能需要另一个端口号。
- (5) 添加服务引用后，再次生成一个客户代理类。给按钮添加一个单击处理程序，在该处理程序中输入如下代码：



```
protected void Button1_Click(object sender, EventArgs e)
{
    var client = new WebServicesSample.SampleServiceSoapClient();
    TextBox2.Text = client.ReverseString(TextBox1.Text);
}
```

代码段 ASPNETClient/Default.aspx.cs

- (6) 生成该项目。选择 `Debug | Start Without Debugging`，可以启动浏览器，在第一个文本框中输入一个测试消息。单击按钮时，会调用 Web 服务，在第二个文本框中得到逆序的消息，如图 19-16 所示。在多项目的解决方案中，必须把启动项目设置为想要启动的项目。

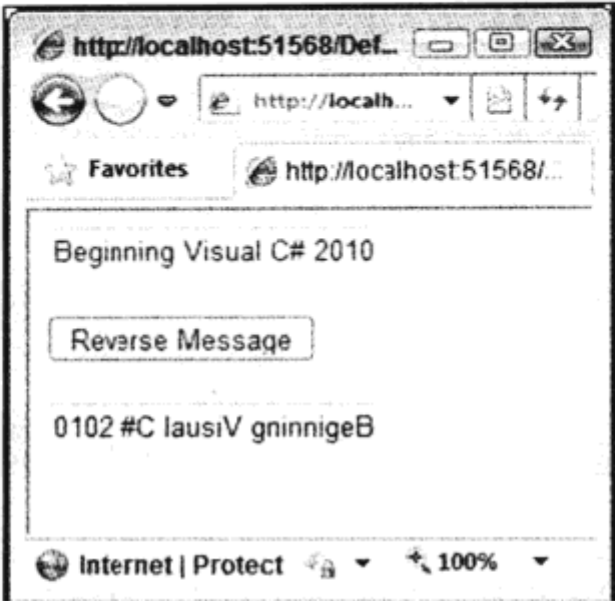


图 19-16

示例的说明

Windows 应用程序中代理类的功能与 Web 应用程序的一样。添加服务引用会创建基于 WSDL 文档的代理类。代理类向服务发送 SOAP 请求。


19.10 传送数据

在前面开发的简单 Web 服务中，只把一个简单的字符串传送给 Web 服务。下面要添加一个方法，从 Web 服务中请求天气信息。这个信息需要在 Web 服务中传送和接收较复杂的数据。

试一试：用 Web 服务传送数据

(1) 使用 Visual Studio 打开前面创建的 Web 服务项目 WebServicesSample。在这个 Web 服务中，用下面的代码定义类型。GetWeatherRequest 类和 GetWeatherResponse 类(参见下面的代码)定义了要在 Web 服务中传送和接收的文档。在这两个类中将使用枚举 TemperatureType 和 TemperatureCondition。

ASP.NET Web 服务使用 XML 序列化，把对象转换为 XML 表示。可以使用名称空间 System.Xml.Serialization 中的特性来决定生成的 XML 格式。



可从  
wrox.com  
下载源代码

```
public enum TemperatureType
{
    Fahrenheit,
    Celsius
}
public class GetWeatherRequest
{
    public string City { get; set; };
    public TemperatureType TemperatureType { get; set; };
}

public enum TemperatureCondition
{
```

代码段 WebServiceSample/GetWeatherRequest.cs

```

        Rainy,
        Sunny,
        Cloudy,
        Thunderstorm
    }

    public class GetWeatherResponse
    {
        public TemperatureCondition Condition { get; set; };
        public int Temperature { get; set; };
    }

```

---

代码段 WebServiceSample/GetWeatherResponse.cs

---

## (2) 添加 Web 服务方法 GetWeather():

```

[WebMethod]
public GetWeatherResponse GetWeather(GetWeatherRequest req)
{
    var resp = new GetWeatherResponse();
    var r = new Random();

    int celsius = r.Next(-20, 50);

    if (req.TemperatureType == TemperatureType.Celsius)
        resp.Temperature = celsius;
    else
        resp.Temperature = (212 - 32) / 100 * celsius + 32;

    if (req.City == "Redmond")
        resp.Condition = TemperatureCondition.Rainy;
    else
        resp.Condition = (TemperatureCondition)r.Next(0, 3);
    return resp;
}

```

---

代码段 WebServiceSample/SampleService.asmx.cs

---

这个方法接收用 `GetWeatherRequest` 定义的数据，返回用 `GetWeatherResponse` 定义的数据。在实现代码中，返回一个随机的天气条件——Microsoft 总部所在地 Redmond Washington 是例外，那里整个星期都在下雨。要生成随机的天气，可使用 `System` 名称空间中的 `Random` 类。

(3) 在生成 Web 服务之后，使用 Windows Forms Application 模板创建一个新项目，把应用程序命名为 `WeatherClient`。

(4) 修改主对话框，如图 19-17 所示。`Temperature Type` 控件中有两个单选按钮，用于选择温度类型(`Celsius` 或 `Fahrenheit`)，城市名可以输入。单击 `Get Weather` 按钮，会调用 Web 服务，Web 服务的结果显示在 `Weather Condition` 和 `Temperature` 文本框控件中。

表 19-5 中列出了控件、控件名称以及 `Text` 属性值。

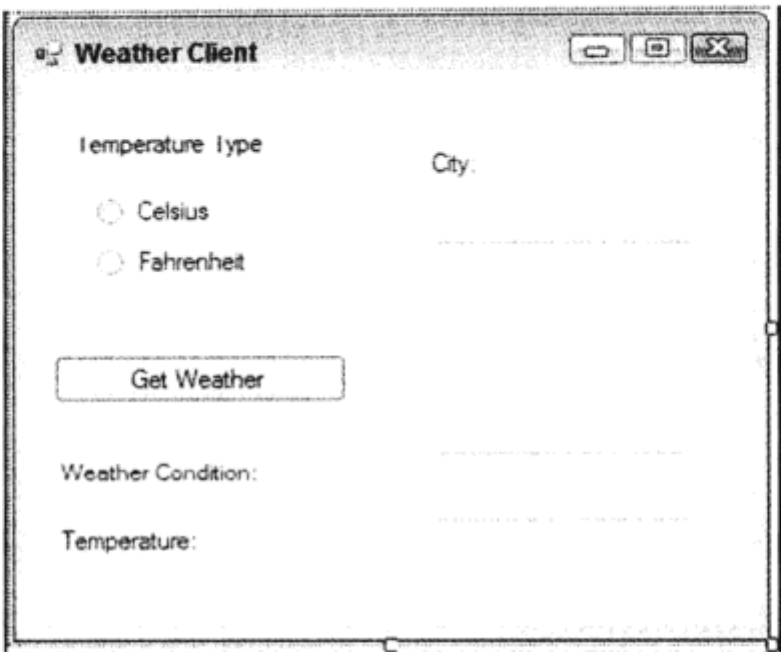


图 19-17

表 19-5

控 件	名 称	Text 属 性
GroupBox	groupBox1	Temperature Type
RadioButton	radioButtonCelsius	Celsius
RadioButton	radioButtonFahrenheit	Fahrenheit
Label	labelCity	City
TextBox	textCity	
Button	buttonGetWeather	Get Weather
Label	labelWeatherCondition	Weather Condition
Label	labelTemperature	Temperature
TextBox	textWeatherCondition	
TextBox	textTemperature	

- (5) 添加对 Web 服务的引用，这类似于前面的客户应用程序项目。把引用命名为 Weather Service。
- (6) 为客户应用程序导入 WeatherClient.WeatherService 名称空间。
- (7) 使用 buttonGetWeather 按钮的 Property 对话框，给按钮添加 Click 事件处理程序 OnGetWeather。
- (8) 给 OnGetWeather()方法添加实现代码，如下所示：



```
private void OnGetWeather(object sender, EventArgs e)
{
    var req = new GetWeatherRequest();
    if (radioButtonCelsius.Checked)
        req.TemperatureType = TemperatureType.Celsius;
    else
        req.TemperatureType = TemperatureType.Fahrenheit;
}
```



```
req.City = textCity.Text;

var client = new SampleServiceSoapClient();
GetWeatherResponse resp = client.GetWeather(req);
textWeatherCondition.Text = resp.Condition.ToString();
textTemperature.Text = resp.Temperature.ToString();
}
```

代码段 WeatherClient/Form1.cs

首先创建一个 `GetWeatherRequest` 对象，它定义了发送给 Web 服务的请求。调用 `GetWeather()` 方法，就可以调用 Web 服务。这个方法返回一个 `GetWeatherResponse` 对象，其值显示在用户界面上。

(9) 启动客户应用程序。输入一个城市，单击 `Get Weather` 按钮。如果幸运，会显示真实天气，如图 19-18 所示。

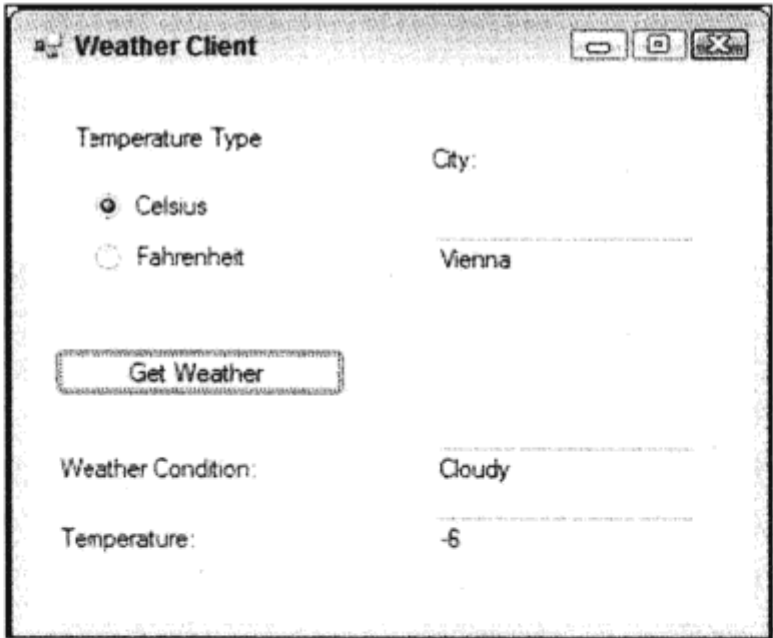


图 19-18

**示例的说明**

在 ASP.NET Web 服务中发送和接收数据需要使用 XML 序列化。通过 XML 序列化，会序列化所有公共属性和公共字段。类 `GetWeatherRequest` 和 `GetWeatherResponse` 使用了公共属性。为了进行 XML 序列化，类必须是公共的，有一个公共的默认构造函数。如果没有给类添加构造函数(与 `GetWeatherRequest` 和 `GetWeatherResponse` 一样)，编译器会默认创建一个公共的默认构造函数，来初始化类的所有成员字段。值类型初始化为 0，引用类型初始化为 `null`。

在 `System.Xml.Serialization` 名称空间中定义的特性类可以用于定制序列化的 XML 结果。`XmlIgnoreAttribute` 特性类用于忽略类成员。可以指定 `XmlElementAttribute` 特性类，来重命名序列化的 XML 元素，使用 `XmlAttributeAttribute` 类可以序列化 XML 特性，而不是元素。

使用带服务选项的 XML 可序列化类型，把 XML 模式信息添加到 WSDL 文档中，此类信息用于在添加服务引用时创建客户应用程序的类型。

### 19.11 小结

本章介绍了 Web 服务，并概述了 Web 服务使用的协议。要查找和运行 Web 服务，必须使用下

述一个或所有的方式：

- 描述 —— WSDL 描述了方法和参数
- 调用 —— 利用 SOAP 协议调用独立于平台的方法

利用 Visual Studio 创建 Web 服务是非常简单的，其中 `WebService` 类使用 `WebMethod` 特性定义了一些方法。创建使用 Web 服务的客户程序与创建 Web 服务一样简单，只需给客户项目添加一个 Web 引用，同时使用代理程序即可。客户程序的核心是 `SoapHttpClientProtocol` 类，它可以把方法调用转换为一个 SOAP 消息。所创建的客户代理提供了异步和同步方法。在使用异步方法时，客户接口不会在 Web 服务方法执行完毕之前停止运行。我们还学习了如何创建定制类，以便定义要传输的复杂数据。第 22 章将介绍如何部署 Web 应用程序和 Web 服务。

### 19.12 练习

下面的练习将使用本章的知识，创建一个新的 Web 服务，为电影院提供座位预定系统。

(1) 创建一个新的 Web 服务 `CinemaReservation`。

(2) 需要 `ReserveSeatRequest` 和 `ReserveSeatResponse` 类来定义在 Web 服务中发送和接收的数据。`ReserveSeatRequest` 类需要一个 `string` 类型的成员 `Name` 来发送名称，还需要两个 `int` 类型的成员，对使用 `Row` 和 `Seat` 定义的座位发送请求。`ReserveSeatResponse` 类定义了发回给客户端的数据，即预定的名称和实际预定的座位。

(3) 创建一个 Web 方法 `ReserveSeat`，它需要把 `ReserveSeatRequest` 作为参数，返回一个 `ReserveSeatResponse`。在 Web 服务的实现代码中，可以使用 `Cache` 对象(参见第 18 章)存储已预定的座位。如果请求的座位还未预定，就返回该座位，把它保存在 `Cache` 对象中。如果请求的座位已被预定，就使用下一个未预定的座位。对于 `Cache` 对象中的座位，可以使用一个二维数组，参见第 5 章。

(4) 创建一个 Windows 客户应用程序，它使用 Web 服务预定电影院中的座位。

附录 A 给出了练习答案。

### 19.13 本章要点

主 题	重 要 概 念
使用 ASP.NET 创建 Web 服务	可以在 ASP.NET Web 项目中创建 ASP.NET Web 服务。使用 <code>WebService</code> 和 <code>WebMethod</code> 特性来定义服务
调用 Web 服务	客户应用程序要调用 Web 服务的操作，可以在 Solution Explorer 中选择 Add   Service Reference，来创建一个代理。添加服务引用，就可以使用 WSDL，创建代理类
异步调用 Web 服务	使用服务引用中的高级选项，可以创建异步方法，以异步方式调用 Web 服务。带 <code>Async</code> 前缀的方法接受 Web 服务方法的输入参数。服务调用结束时，会引发一个事件，接收输出参数
在 Web 服务之间传送数据	要把非简单的数据类型传送给 Web 服务，可以创建一个定制类。XML 序列化用于把对象转换为在线路上传送的消息

# 第 20 章

## 部署 Web 应用程序

### 本章内容:

---

- 如何为 ASP.NET Web 应用程序配置 IIS
- 如何复制 Visual Studio Web 站点
- 如何发布 Web 应用程序
- 如何为 Web 应用程序创建 Windows 安装程序包

前面 2 章学习了如何用 ASP.NET 开发 Web 应用程序和 Web 服务。对于这些类型的应用程序，有不同的部署选项。可以复制 Web 页面，发布 Web 站点，或者创建安装程序。本章介绍这些不同选项的优缺点，以及如何完成这些任务。

### 20.1 Internet Information Services

对于用 Visual Studio 2010 开发的 Web 应用程序，不需要安装 Internet Information Services (IIS)。因为 Visual Studio 2010 有自己的 Web 服务器：Visual Web Development Server。这是一个简单的 Web 服务器，只运行在本地机器上。在产品系统上，必须有 IIS 才能运行 Web 应用程序。

IIS 不能用于 Windows 7 Home Edition。在其他版本上，可以安装 IIS，其方式与安装其他 Windows 组件相同。在控制面板上，单击 Programs。其中包含一个类别 Programs and Features 和一个链接 Turn Windows features on or off。单击这个链接。Windows 的一个功能是 Internet Information Services，需要选择它才能安装 Internet Information Services。还可以要求系统管理员在系统上安装 IIS。

只有用 IIS 配置 ASP.NET 运行库，才能运行 ASP.NET Web 应用程序。只要用 IIS 管理器工具检查处理程序映射(如图 20-1)，就可以验证 ASP.NET 运行库是否配置。如果安装了 IIS，这个工具就位于 Administrative Tools 中，其菜单项是 Internet Information Services (IIS) Manager。如果没有把 Administrative Tools 配置为可以直接从 Start 菜单中使用，就可以在控制面板上查找 Administrative Tools。

在 Internet Information Services (IIS) Manager 中，双击 Handler Mappings。在信息中滚动，可以

看到\*.aspx 路径配置了多次。可以找到.NET Framework 的多个版本, 还有本机配置和托管配置。.aspx 扩展名的 IsapiModule 配置定义了本机配置 System.Web.UI.PageHandlerFactory, 这是处理请求的.NET 类。

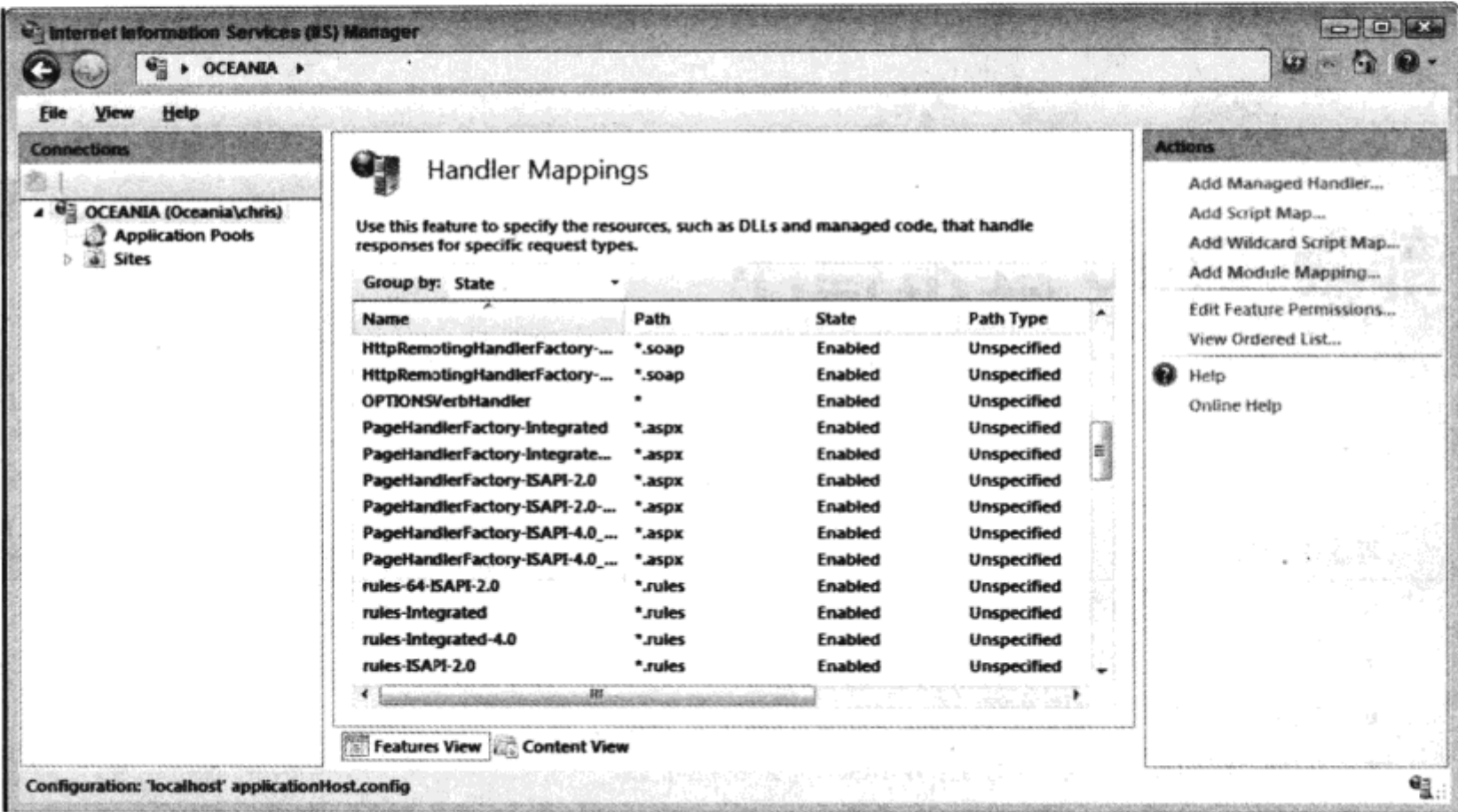


图 20-1

如果没有在系统上配置 ASP.NET 运行库的处理程序映射, 可以启动程序 `aspnet_regiis -i`, 使用 IIS 安装文件扩展名和模块。

IIS 的主进程是 `inetinfo.exe`, 它运行在权限很高的 System 账户下。配置了 IsapiModule 后, 就把对 ASPX 文件的请求发送给辅助进程(`w3wp.exe`)。不同的辅助进程可以配置为运行.NET 运行库的不同版本。也可以配置用户身份, 以该用户身份运行这个进程, 指定循环选项。

## 20.2 IIS 配置

必须在运行 Web 应用程序之前配置 IIS。下面的示例将用 Internet Information Services Manager 工具创建一个 Web 站点, 首先, Web 站点需要一个虚拟目录, 这是客户端用于访问 Web 应用程序的目录。例如 `http://server/mydirectory`, `mydirectory` 就是一个虚拟目录。虚拟目录完全独立于在磁盘上存储文件的物理目录。例如, `mydirectory` 的物理目录是 `D:\someotherdirectory`。

试一试: 创建新的应用程序池

- (1) 启动 Internet Information Services Manager 工具, 这个工具位于控制面板的管理工具上。如图 20-2 所示。
- (2) 在树型视图中选择 Application Pools, 右击它, 从关联菜单中选择 Add Application Pool 选项。

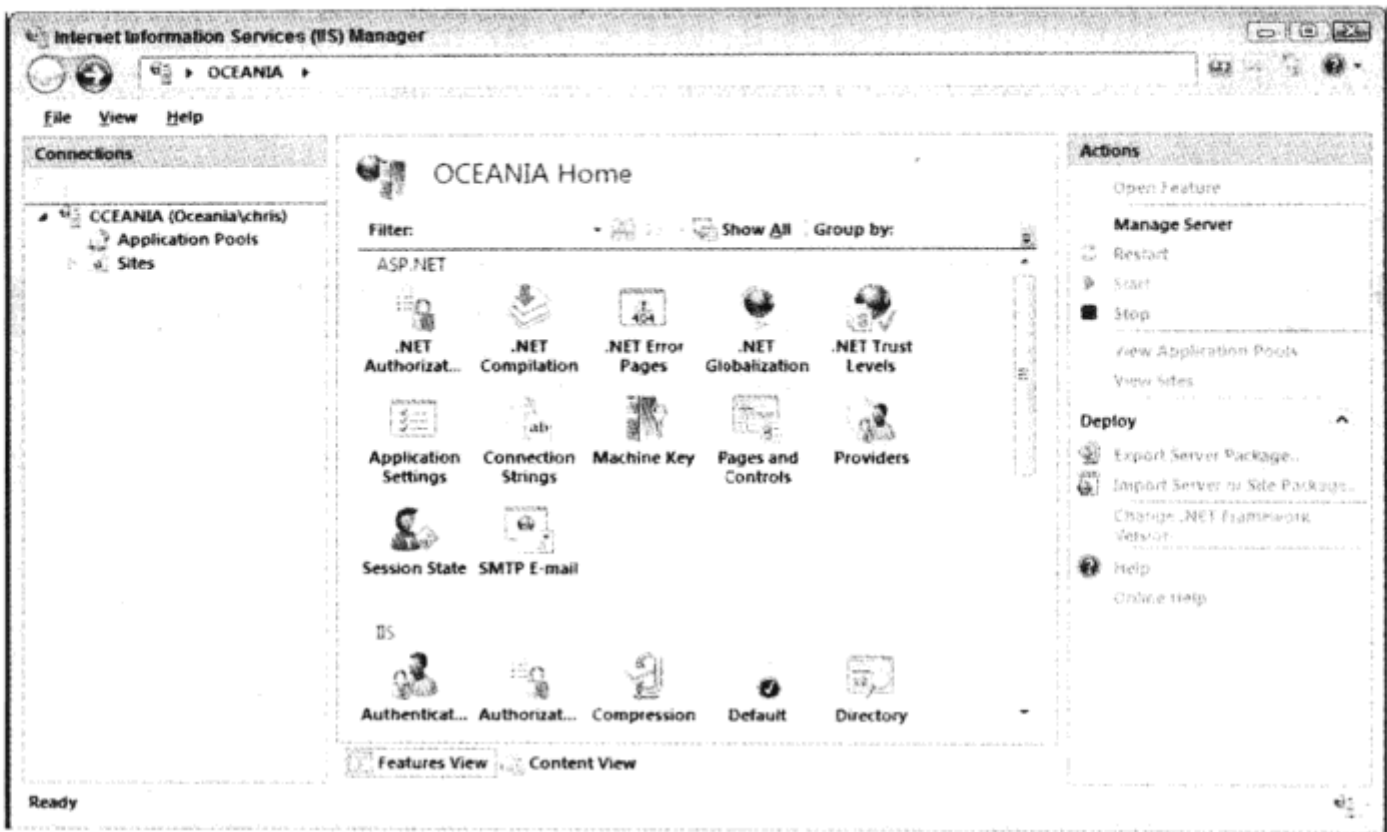


图 20-2

(3) 打开 Add Application Pool 对话框, 如图 20-3 所示。在 Name 文本框中, 输入 Beginning Visual C# App Pool, 再选择 .NET Framework version v4.0。单击 OK 按钮。实际上, 我们配置了 .NET 运行库的版本。对于 ASP.NET 2.0、3.0 和 3.5 应用程序, 可以使用相同的版本号 2.0.50727。

(4) 创建应用程序池后, 就可以配置高级设置了, 如图 20-4 所示。定义进程在什么身份下运行, 在多核或多 CPU 系统上还要指定应使用哪些 CPU, 并定义在这个池中应运行的辅助进程数。从 Internet Information Services (IIS) Manager 右边的 Actions 类别中选择了应用程序池, 或从关联菜单中选择了它后, 就可以配置高级设置了。

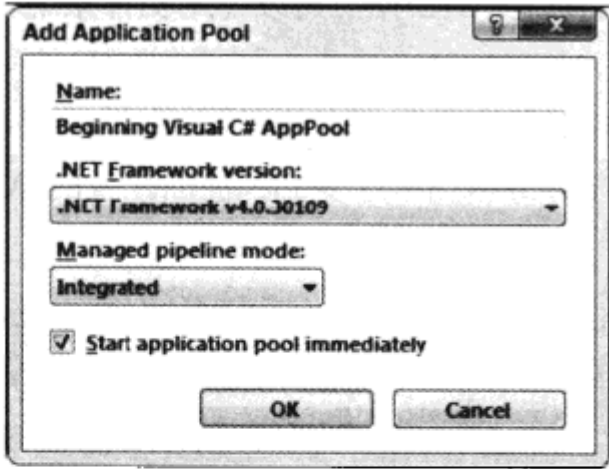


图 20-3

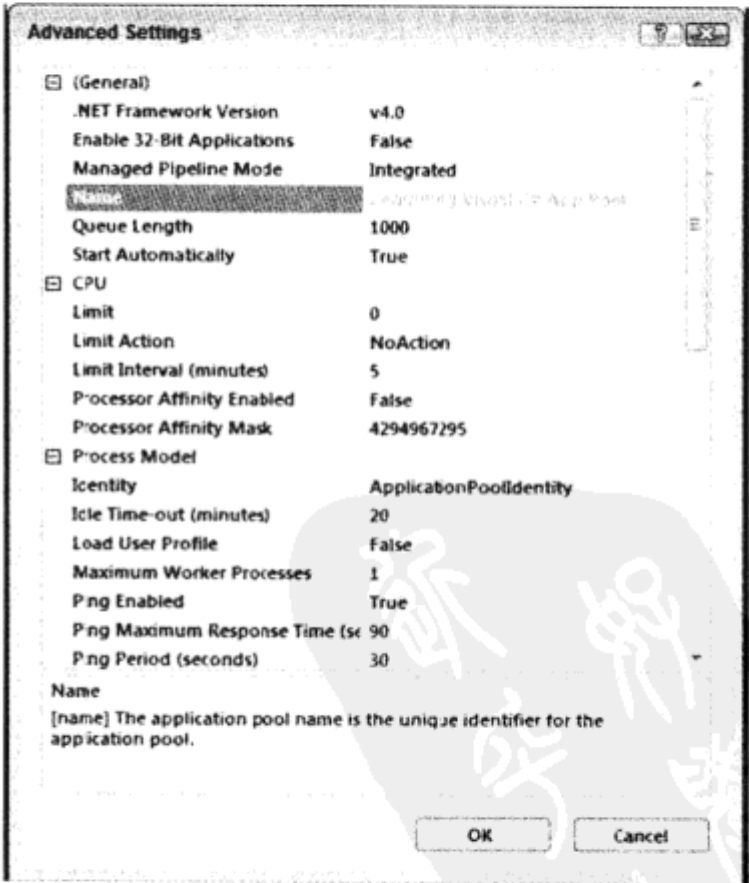


图 20-4



示例的说明

应用程序池允许不同的 Web 站点运行 ASP.NET 运行库的不同版本，可以有不同的用户账户和稳定性。

配置好应用程序池后，就可以创建新的 Web 应用程序了，如下面的示例所示。

试一试：创建新的 Web 应用程序

- (1) 在 IIS Manager 的树型视图中，选择 Default Web Site。
- (2) 右击后从关联菜单中选择 Add Application 选项。打开 Add Application 对话框，如图 20-5 所示。

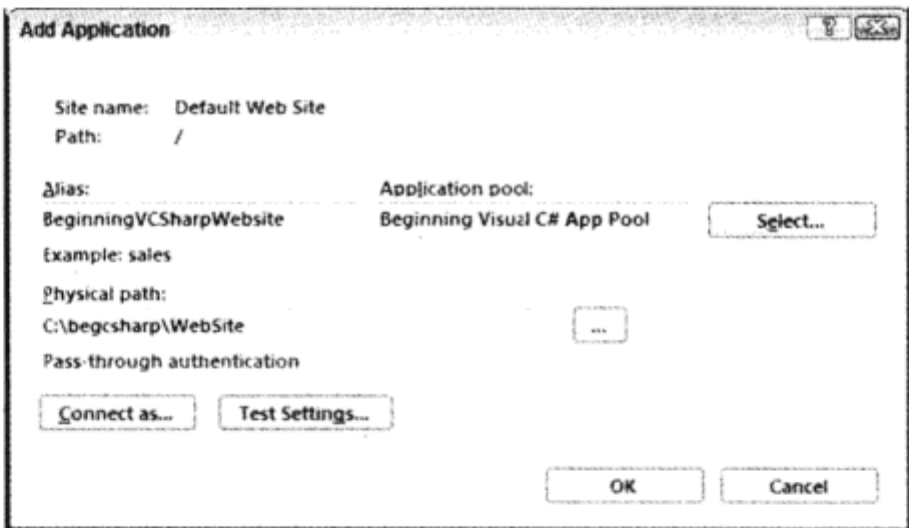


图 20-5

- (3) 输入 Web 站点的物理路径和别名 BeginningVCSharpWebsite。选择刚才创建的应用程序池 Beginning Visual C# App Pool。
  - (4) 单击 OK 按钮。
- 配置好虚拟目录后，就可以在 VS 中将 Web 应用程序复制或发布到这个 Web 站点了。

20.3 复制 Web 站点

在 Visual Studio 2010 中，可以把文件从源 Web 站点复制到远程 Web 站点上。源 Web 站点就是用 Visual Studio 打开 Web 应用程序的 Web 站点，这个 Web 站点可以从本地文件系统或 IIS 上访问，具体取决于 Web 应用程序的创建方式。文件复制目的地的远程 Web 站点可以使用文件系统、FTP 协议或 IIS 上的 FrontPage Server Extensions 访问。

复制文件可以双向进行：从源 Web 站点复制到远程 Web 站点上，或者从远程 Web 站点复制到源 Web 站点上。下面的示例将使用 Visual Studio 把新建的 Web 应用程序复制到前面配置的 Web 站点上。

复制网站的 VS 菜单只能用于 Web Site，不能用于 Web Project.

试一试：复制 Web 站点

- (1) 在 Windows 7 或 Windows Vista 中，用提升的管理权限启动 Visual Studio 2010。把 Web 站点复制到本地 IIS 上需要管理权限。在 Windows XP 上，如果用管理权限登录到一个账户上，就可以正常启动 Visual Studio。
- (2) 选择 File | New Web Site 菜单，再选择 ASP.NET Web Site 模板，创建一个新的 Web 站点。把本地文件系统选择为这个 Web 站点的位置。这会创建包含几个页面和样式的示例站点。
- (3) 选择菜单项 Website | Copy Web Site，打开如图 20-6 所示的对话框。

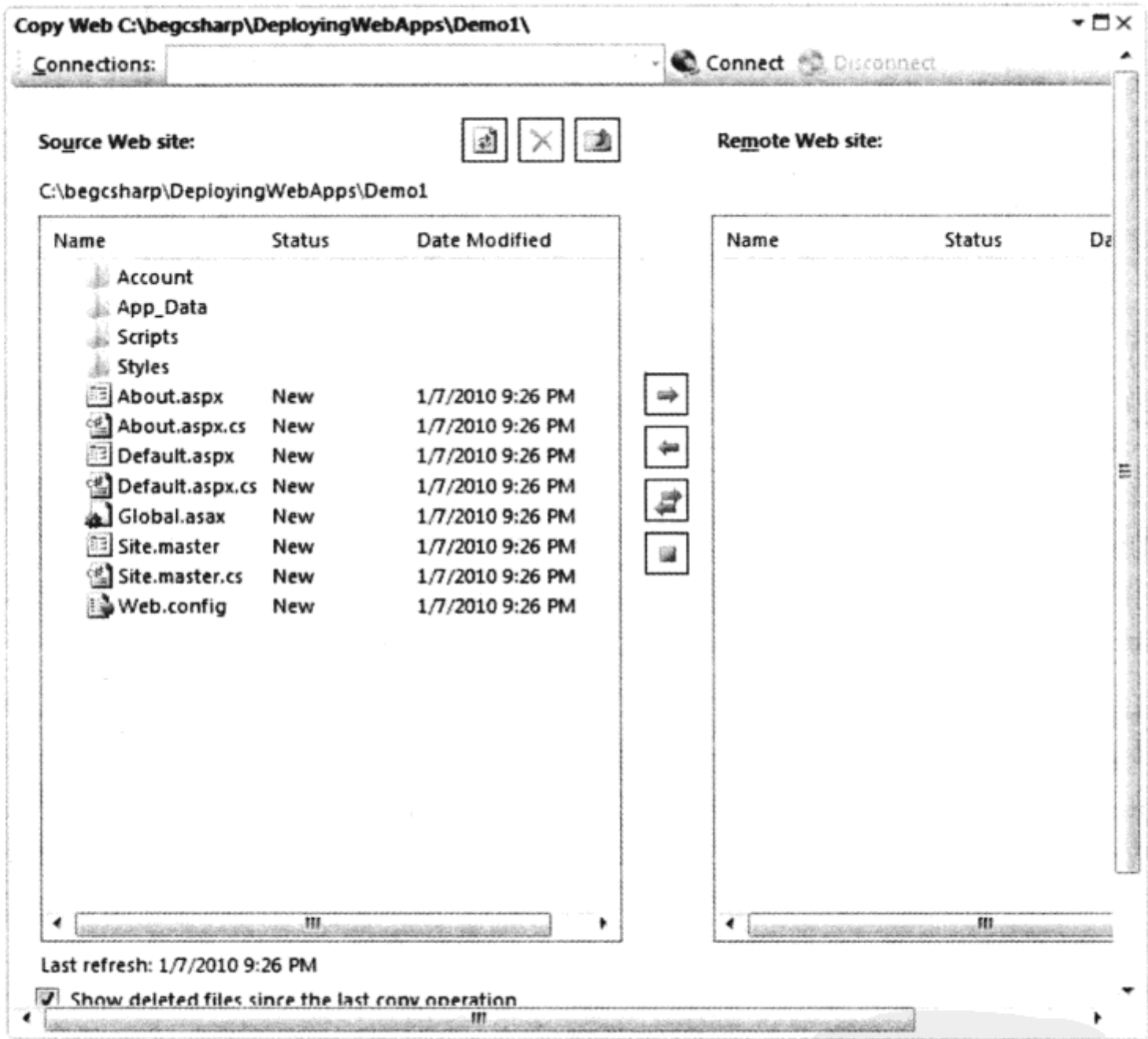


图 20-6

- (4) 单击窗口顶部的 Connect 按钮，打开如图 20-7 所示的 Open Web Site 对话框。
- (5) 在 Open Web Site 对话框中，可以选择要复制到本地文件系统、本地 IIS、FTP 站点和远程站点(已安装了 FrontPage Server Extensions)中的文件。选择 Local IIS，再选择前面创建的 Web 站点 BeginningVCSharpWebsite。如果运行的是 Windows Home Edition，就只能把文件复制到本地文件系统上，因为不能使用 IIS。



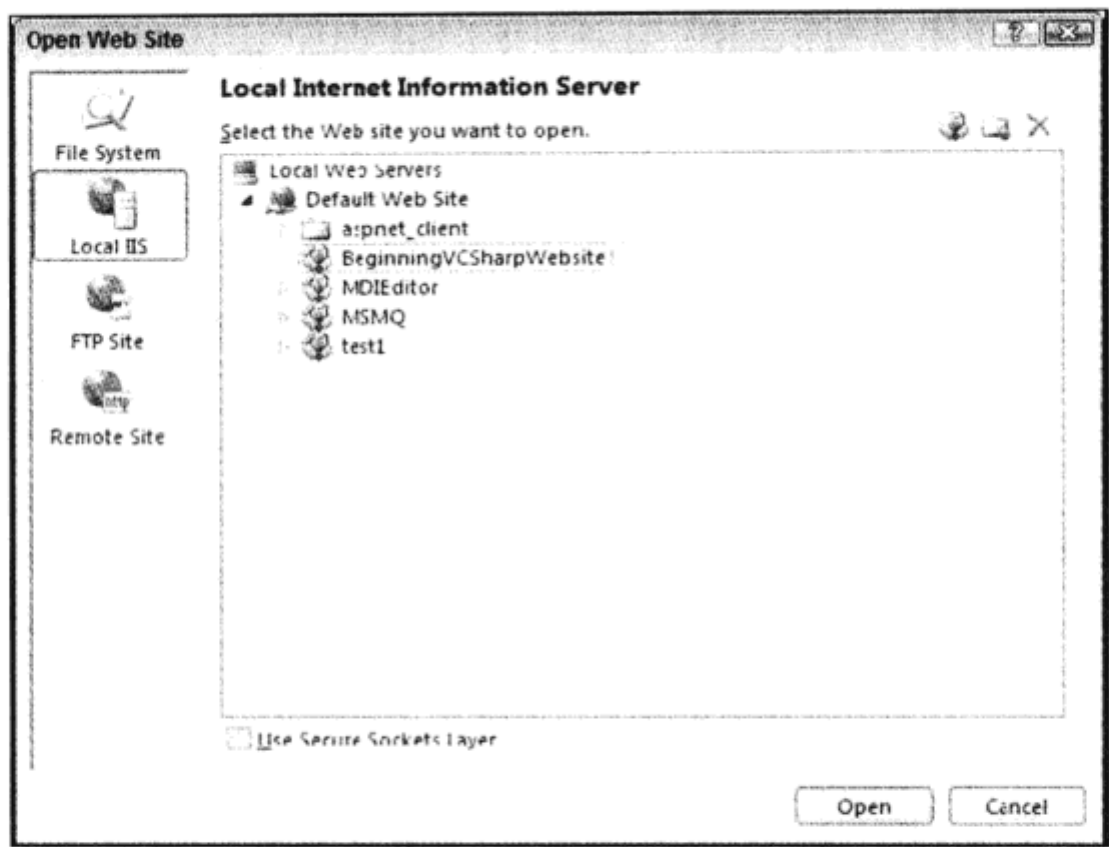


图 20-7

- (6) 在 Source Web site 列表中，选择要从 Source Web Site 复制到 Remote Web Site 的文件。
- (7) 单击 Copy Selected Files 按钮。这个按钮在 Source Web Site 视图和 Remote Web Site 视图的中间，有一个箭头。如果把鼠标停放在按钮上，就会显示一个工具提示，解释按钮的用途。箭头方向表示文件的复制方向，从源站点复制到目标站点，或者从目标站点复制到源站点。箭头指向两个方向的按钮验证哪些文件是新文件，把新文件复制到源站点和目标站点上。
- (8) 现在所有选中的文件都已被复制到新的 Web 站点上。可以打开浏览器，输入链接 <http://localhost/BeginningVCSharpWebsite>，就会看到复制的 Web 站点。

示例的说明

使用 Copy Web Site 工具，还可以选择要从远程 Web 站点复制到源 Web 站点的文件。单击 Synchronize Selected Files 显示双向按钮，新文件就会从远程 Web 站点复制到源 Web 站点上，也会从源 Web 站点复制到远程 Web 站点上。如果其他开发人员在同一个公用 Web 服务器上同步文件，这就是一个很有用的选项。在两个方向上同步，会把新文件复制到公用 Web 服务器上，同时把同事的远程 Web 服务器上的文件复制到本地站点上。

复制文件时，不能确保是否能编译文件。编译应在浏览器访问文件时进行。使用命令行工具 aspnet\_compiler.exe 就可以对 Web 站点进行预编译。

如果输入命令

```
aspnet_compiler -v /BegVCSharpWebsite
```

就预编译了 Web 站点 BeginningVCSharpWebsite。这样第一个用户就不必等到 ASPX 页面编译后才访问 Web 站点了。

这个工具位于 .NET 运行库的目录下。

## 20.4 发布 Web 站点

使用 Visual Studio 2010 Web 项目，还可以发布 Web 应用程序。如果没有自包含的 IIS，且需要把 Web 应用程序发布给提供商，这就是最佳选项。

Visual Studio 2010 中的几个发布选项如下所示：

- 发布到文件系统上
- 发布到安装了 FrontPage Server Extensions 的服务器上
- 使用 FTP
- 使用 1-Click 发布选项，这是 Visual Studio 2010 的一个新特性。1-Click 选项只能用于支持该特性的主机上，但这样的主机很多，很容易就能找到

下面的示例使用 Visual Studio 的新发布特性发布 Web 应用程序。

试一试：发布 Web 应用程序

- (1) 打开第 18 章创建的 Web 项目 EventRegistrationWeb。
- (2) 打开 Package/Publish 项目设置，如图 20-8 所示。选择创建发布包的位置。单击 Include all Databases configured in Deploy SQL Tab 设置旁边的链接 Open Settings。

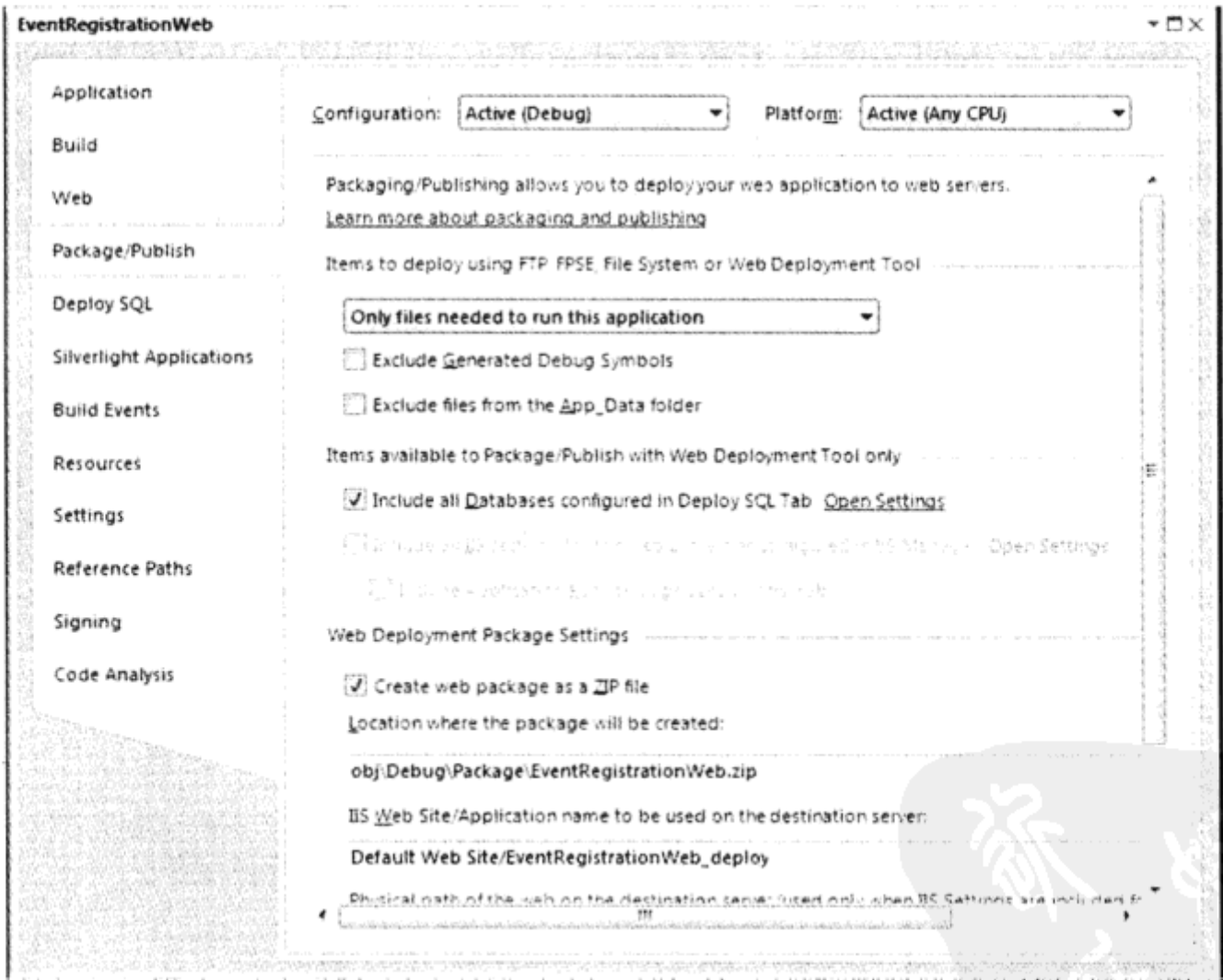


图 20-8

- (3) 显示 Deploy SQL 设置，如图 20-9 所示。单击 Import from Web.config 按钮，导入数据库连接字符串，就可以部署该连接字符串引用的数据库。验证其他设置。可以定义指向目标数据库服务器的连接字符串，数据库数据和模式应写入数据库服务器中。

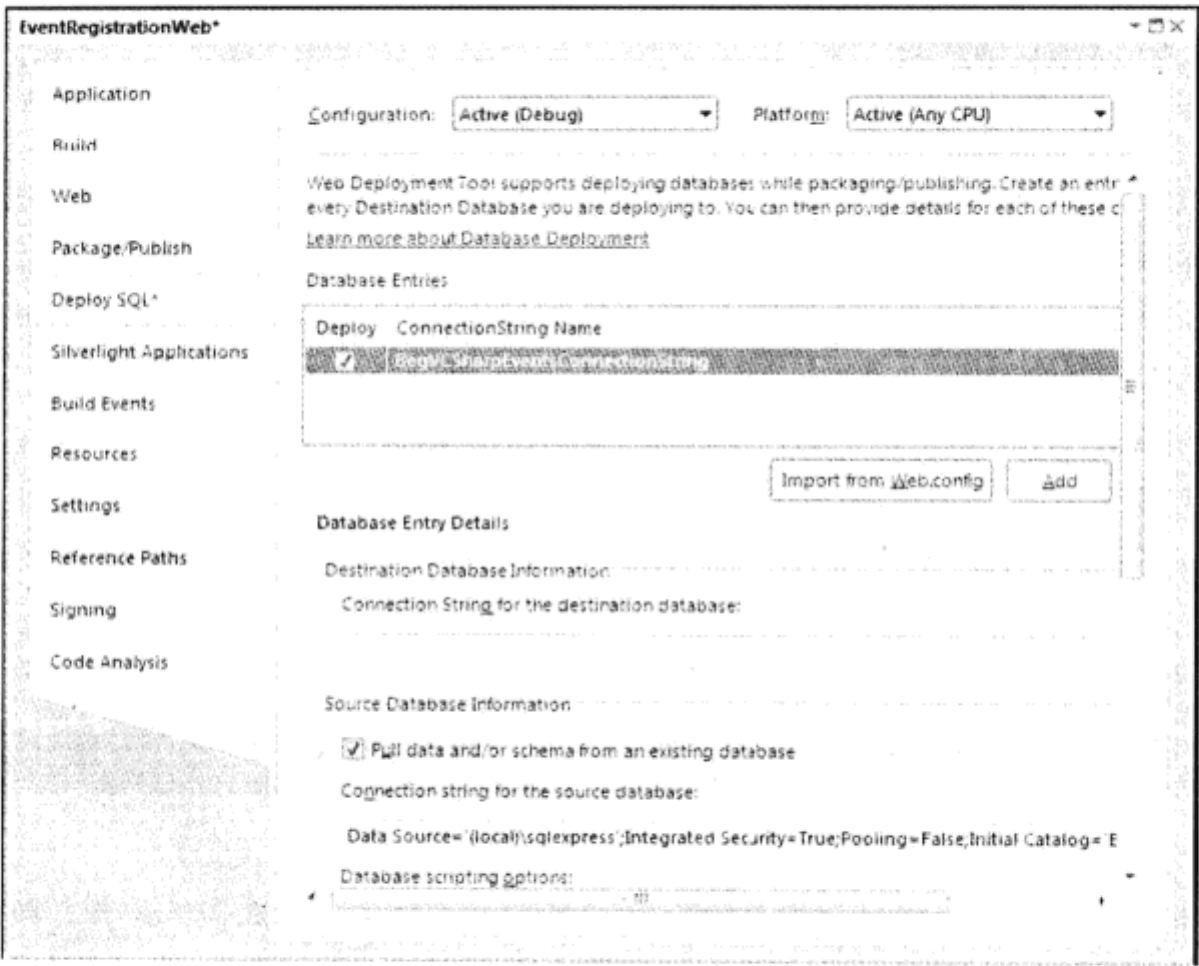


图 20-9

(4) 在 Visual Studio Build 菜单中，选择 Publish 菜单项，打开如图 20-10 所示的 Publish Web 对话框。选择发布方法 MSDeploy.Publish 的设置。这是一个 1-Click 发布选项，可用于几个主机提供商。单击链接 Click Here，在当地找到一个主机公司，如果不使用这个发布选项，可以把发布方法改为 File System。当然，如果使用支持这个发布选项的主机提供商，就可以使用它发布 Web 应用程序。

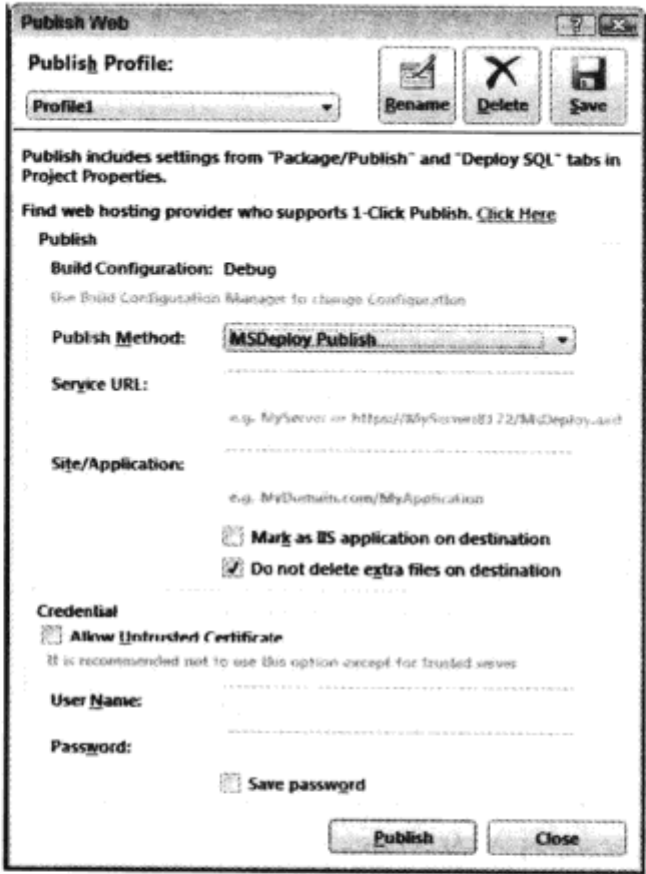


图 20-10

(5) 选择发布到文件系统上后，会打开如图 20-11 所示的对话框。在 Target Location 设置中输入一个本地目录，单击 Publish 按钮。

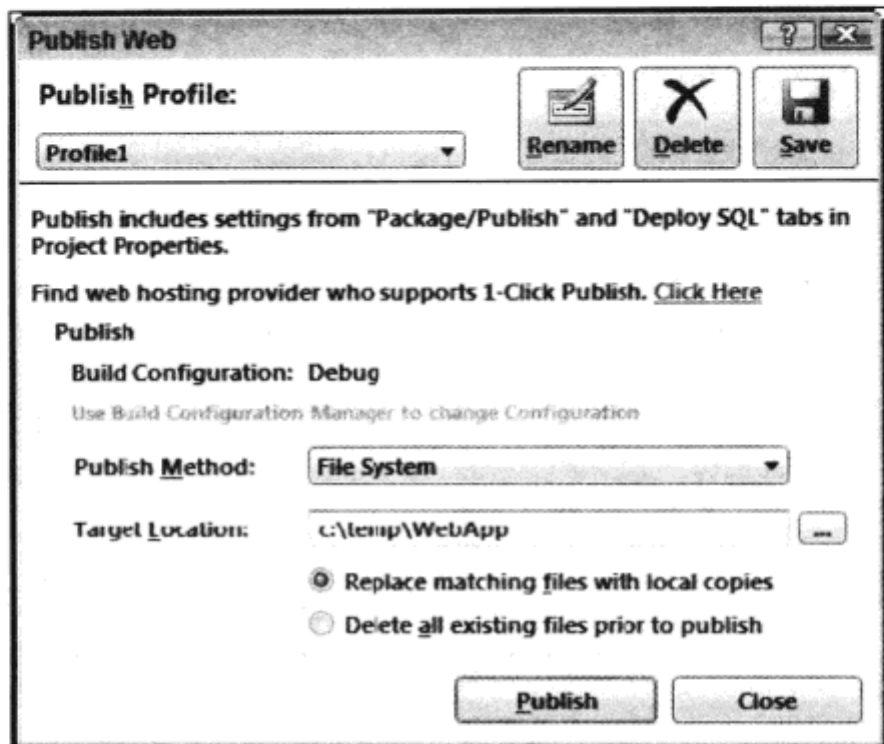


图 20-11

(6) 使用 Windows 资源管理器验证目标位置，检查已发布的文件。

## 20.5 Windows 安装程序

也可以创建一个 Windows 安装程序来安装 Web 应用程序。如果 Web 应用程序需要共享的程序集，就需要创建安装程序。使用安装程序的优点在于，虚拟目录用 IIS 配置，不需要手工创建虚拟目录。安装 Web 应用程序的用户可以启动 setup.exe 程序，安装过程会自动进行。当然，启动这个程序需要管理权限。

### 20.5.1 创建安装程序

Visual Studio 2010 提供了项目类型 Web Setup Project，用于为 Web 应用程序创建安装程序。通过 Web Setup Project，可以使用下列编辑器：File System、Registry、File Types、User Interface、Custom Action 和 Launch Conditions。第 17 章在讨论 Windows 应用程序时已讨论过它们。这里只介绍与 Web 应用程序相关的编辑器。

下面的示例将创建一个安装程序，可供安装 Web 应用程序。

#### 试一试：创建安装程序

- (1) 用 Visual Studio 2010 打开第 18 章的 Web 应用程序 EventRegistrationWeb。
- (2) 在同一个解决方案中添加一个 Web Setup Project 类型的新项目，如图 20-12 所示，把该项目命名为 EventRegistrationWebSetup，单击 OK 按钮。

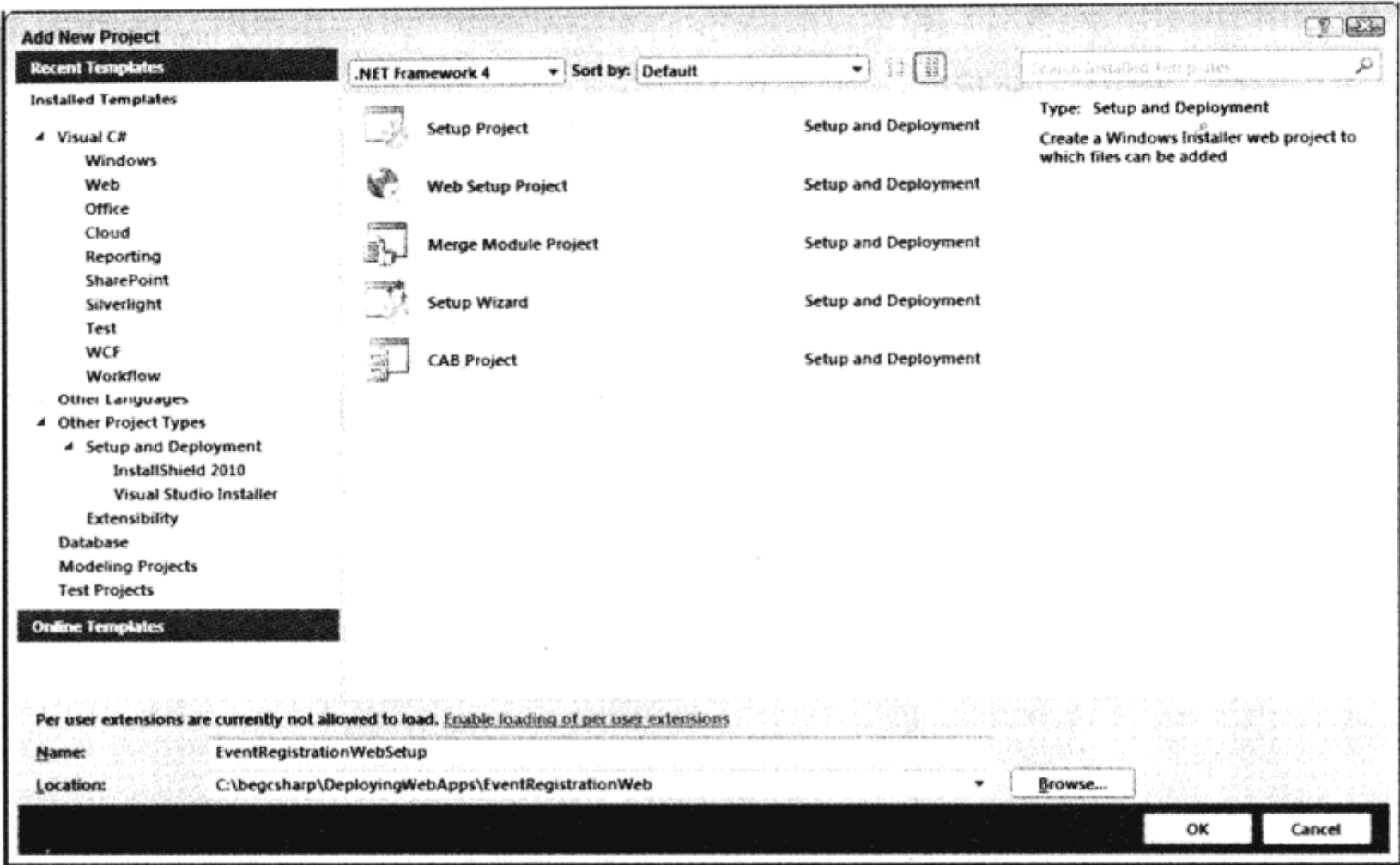


图 20-12

(3) 如果创建项目后没有打开 File System 编辑器，就打开它。选择 File System on Target Machine，再选择菜单项 Project | Add | Project Output，在 Project Output 对话框中选择 Content Files of the Web application，单击 OK 按钮。

(4) 在 File System 编辑器中，选择 Web Application Folder。现在可以用属性编辑器配置 Web 应用程序了。表 20-1 中列出了属性及其说明。

表 20-1

Web Application Folder 属性	说 明
AllowDirectoryBrowsing	一个 IIS 配置选项。把这个选项设置为 true，就可以浏览 Web 站点上的文件，其默认值是 false
AllowReadAccess	这个选项默认设置为 true。要访问 ASPX 页面，需要拥有读取访问权限
AllowScriptSourceAccess	默认情况下，把这个属性设置为 false，拒绝对脚本源文件的访问
AllowWriteAccess	默认为拒绝写入访问
DefaultDocument	设置 Web 站点的主页，如 Default.aspx
ExecutePermissions	默认设置为 vsdepScriptsOnly，它允许访问 ASP.NET 页面，但不允许在服务器上运行定制的可执行文件。如果允许在服务器上运行定制的可执行文件，这个选项就可以设置为 vsdepScriptsAndExecutables
LogVisits	设置为 true，就配置客户访问日志
VirtualDirectory	设置用 IIS 配置的虚拟目录名

(5) 用菜单项 View | Editor | Launch Conditions 打开 Launch Conditions 编辑器。安装条件定义了在安装进行之前必须安装在目标系统上的产品。

(6) 检查已配置的安装条件。Search for IIS. 配置检查注册表项 SYSTEM\CurrentControlSet\Services\W3SVC\Parameters, 获得 IIS 的版本, 就可以验证 IIS 是否安装在目标系统上。使用默认添加的如下条件:

```
(IISMAJORVERSION >= "#5" AND IISMINORVERSION >= "#1") OR IISMAJORVERSION >= "#6"
```

安装条件 IIS Condition 确认 IIS 版本至少是 5.1。

(7) 选择 Build | Build EventRegistrationWebSetup 菜单项, 生成安装应用程序。

(8) 在安装项目的目录下, 有一个 setup.exe 文件和一个安装软件包 EventRegistrationWebSetup.msi。

## 20.5.2 安装 Web 应用程序

启动 setup.exe 程序, 就可以安装 Web 应用程序。

试一试: 安装 Web 应用程序

(1) 单击 setup.exe, 开始安装 Web 应用程序。在 User Account Control 对话框中, 单击 Yes, 允许修改。打开的 Setup Wizard 对话框如图 20-13 所示。单击 Next 按钮。

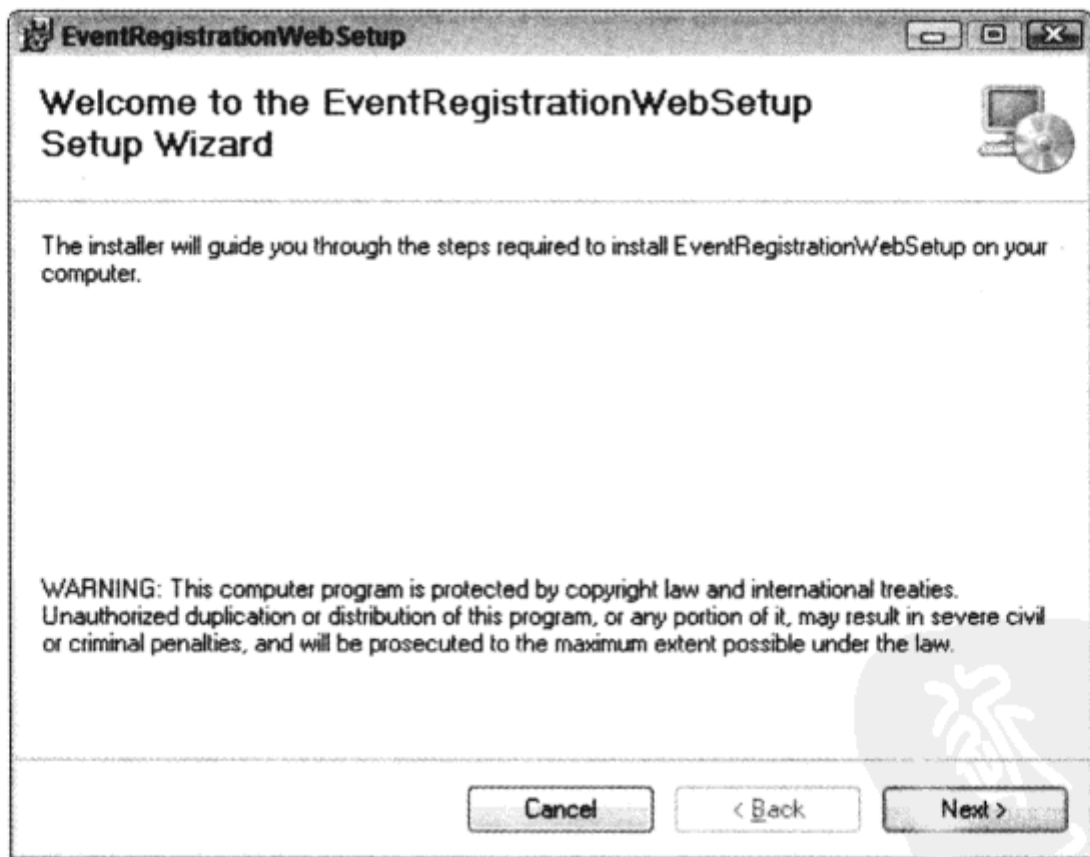


图 20-13

(2) 在 Select Installation Address 页面上, 如图 20-14 所示, 把虚拟目录重命名为没有用 IIS 配置的名称。接着选择前面创建的应用程序池(Beginning Visual C# App Pool), 单击 Next 按钮。

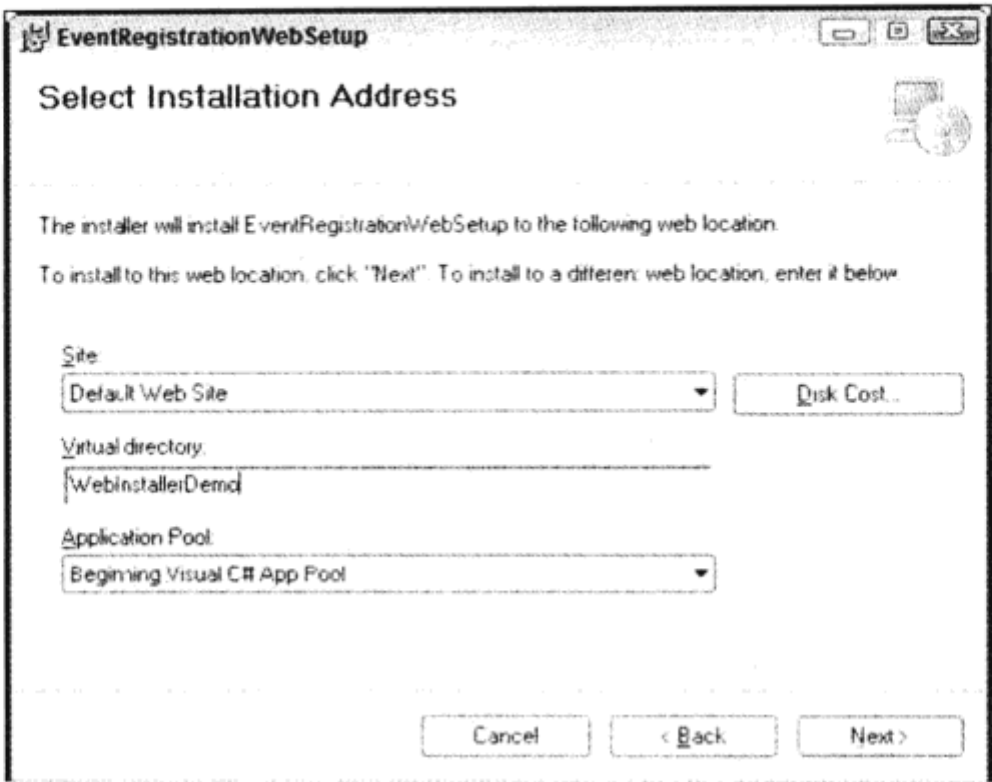


图 20-14

(3) 在 **Confirm Installation** 对话框中单击 **Next** 按钮，确认安装，下一个对话框会在安装程序运行过程中，显示出进度条。

(4) 成功安装后，会显示 **Installation Complete** 对话框，如图 20-15 所示。单击 **Close** 按钮。

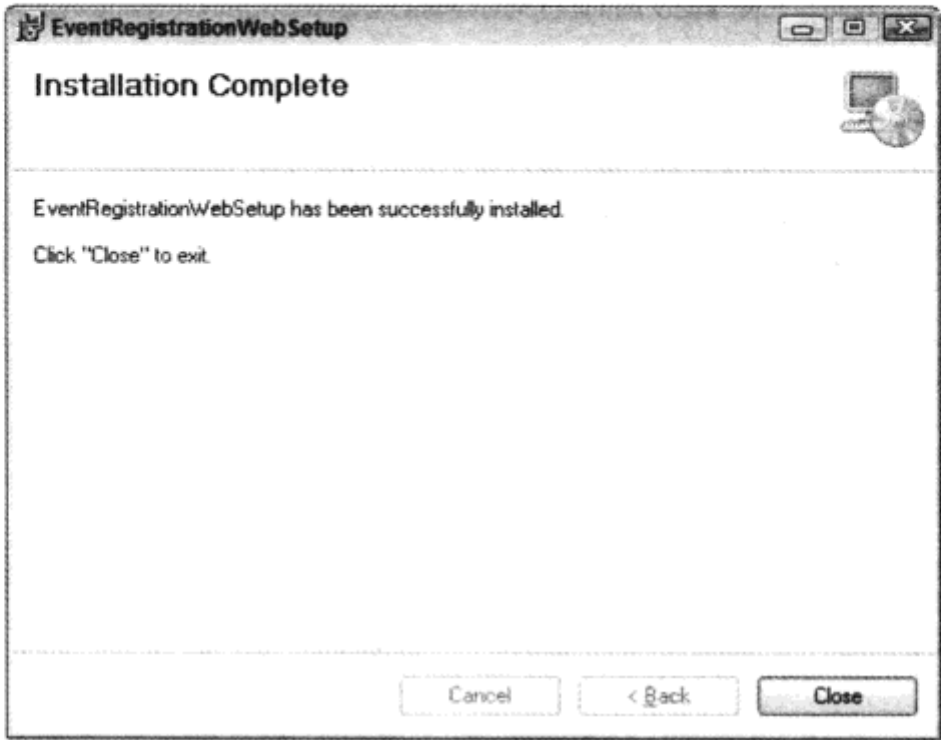


图 20-15

(5) 现在可以在新的虚拟目录下启动 Web 站点了。

## 20.6 小结

本章介绍了部署 Web 应用程序的不同选项。Copy Web Site 工具可以使用文件共享、FTP 或 FrontPage Server Extensions 把文件复制到 Web 服务器上。文件的同步可以双向进行。发布 Web 应用



程序是 VS2010 的一个新特性，可以将其与 1-Click 发布选项一起使用，发布到目录、FTP 服务器或安装了 FrontPage Server Extensions 的 IIS 上。如果有 IIS 上的管理权限进行发布，就可以在 IIS 中创建一个新的应用程序，来安装 Web 应用程序。安装项目不仅复制 ASP.NET 页面和程序集，还在 IIS 中创建了一个虚拟目录。

20.7 练习

- (1) 复制和发布 Web 应用程序有什么区别？在什么场合下使用它们？
  - (2) 何时使用安装程序比复制站点更合适？
  - (3) 发布 Web 项目有哪些不同的选项？各个发布选项有什么要求？
  - (4) 把第 19 章的 Web 服务发布到用 IIS 定义的一个虚拟目录中。
- 附录 A 给出了练习答案。

20.8 本章要点

主 题	重 要 概 念
IIS 配置	为了在 IIS 上运行 ASP.NET Web 应用程序，必须配置 IIS。在请求带有特定扩展名(如.aspx)的文件时，处理程序映射会定义要调用的类。通过应用程序池配置，可以定义要使用的.NET 运行库版本
复制 Web 站点	发布 Web 应用程序的一个简单选项是使用复制功能。复制 Web 站点的菜单不能用于 Web 项目，只能用于 Visual Studio Web 站点。可以在开发计算机和服务端之间来回复制文件
发布 Web 应用程序	如果使用 Web 站点的主机，就可以使用 Visual Studio 2010 中发布 Web 应用程序的一个简单选项 1-Click。使用发布菜单，还可以把 Web 应用程序发布到 FTP 服务器和文件系统上。Web 应用程序使用的数据库也可以发布
Web 应用程序的 Windows 安装程序	如果需要在 IIS 中创建 Web 应用程序，就可以使用安装程序。Web Setup Project 模板会创建一个 Windows 安装程序文件，它不仅会复制 Web 应用程序的内容，还会创建一个虚拟目录





# 第Ⅳ部分

## 数 据 访 问

---

- 第 21 章 文件系统数据
- 第 22 章 XML
- 第 23 章 LINQ 简介
- 第 24 章 应用 LINQ



# 第21章

## 文件系统数据

本章的主要内容:

---

- 流的含义, 以及.NET 如何使用流类访问文件
- 如何使用 File 对象处理文件结构
- 如何读写文件
- 如何在文件中读写格式化的数据
- 如何读写压缩文件
- 如何序列化和反序列化对象
- 如何监控文件和目录的变化

本章将介绍如何读写文件, 这是许多.NET 应用程序的一个基础性工作。我们要讨论用于创建、读写文件的主要类, 支持在 C#代码中处理文件系统的类。本章不详细介绍全部的类, 但将深入介绍一些类, 以便读者理解概念和基本理论。

文件是在应用程序的实例之间存储数据的一种便利方式, 也可以用于在应用程序之间传输数据。文件可以存储用户和应用程序配置, 以便在下次运行应用程序时检索它们。定界的文本文件(例如用逗号分隔的文件)由许多旧系统使用; 为了与这些旧系统进行交互, 还需要了解如何处理定界数据。.NET Framework 提供的工具可以在应用程序中有效地使用文件。

### 21.1 流

在.NET Framework 中进行的所有输入和输出工作都要用到流(stream)。流是序列化设备(serial device)的抽象表示。序列化设备可以以线性方式存储数据, 并可以按同样的方式访问: 一次访问一个字节。此设备可以是磁盘文件、网络通道、内存位置或其他支持以线性方式读写的对象。把设备变成抽象的, 就可以隐藏流的底层目标和源。这种抽象的级别支持代码重用, 允许编写更通用的例程, 因为不必担心数据传输方式的特性。因此, 当应用程序从文件输入流、网络输入流或其他流中读取数据时, 就可以转换并重用类似的代码。而且, 使用文件流还可以忽略每种设备的物理机制,

无需担心硬盘头或内存分配问题。

有两种类型的流：

- **输出流：**当向某些外部目标写入数据时，就要用到输出流。这可以是物理磁盘文件、网络位置、打印机或另一个程序。理解流编程技术可以带来许多高级应用。本章仅讨论文件系统数据，所以只介绍写入磁盘文件。
- **输入流：**用于将数据读入程序可以访问的内存或变量中。到目前为止，我们使用的最常见的输入流形式是键盘。输入流可以来自任何源，在此主要关注读取磁盘文件。适用于读/写磁盘文件的概念也适用于大多数设备，因此通过本章的学习，我们就会对流有基本的认识，并学习可以应用于许多情况的有效方法。

21.2 用于输入和输出的类

System.IO 命名空间包含本章要介绍的几乎所有的类。System.IO 包含用于在文件中读写数据的类，必须在 C#应用程序中引用此名称空间才能访问这些类，而无需完全限定类型名。在 System.IO 命名空间中包含了不少类，在图 21-1 中就可以看到，但这里仅介绍用于文件输入和输出的主要类。

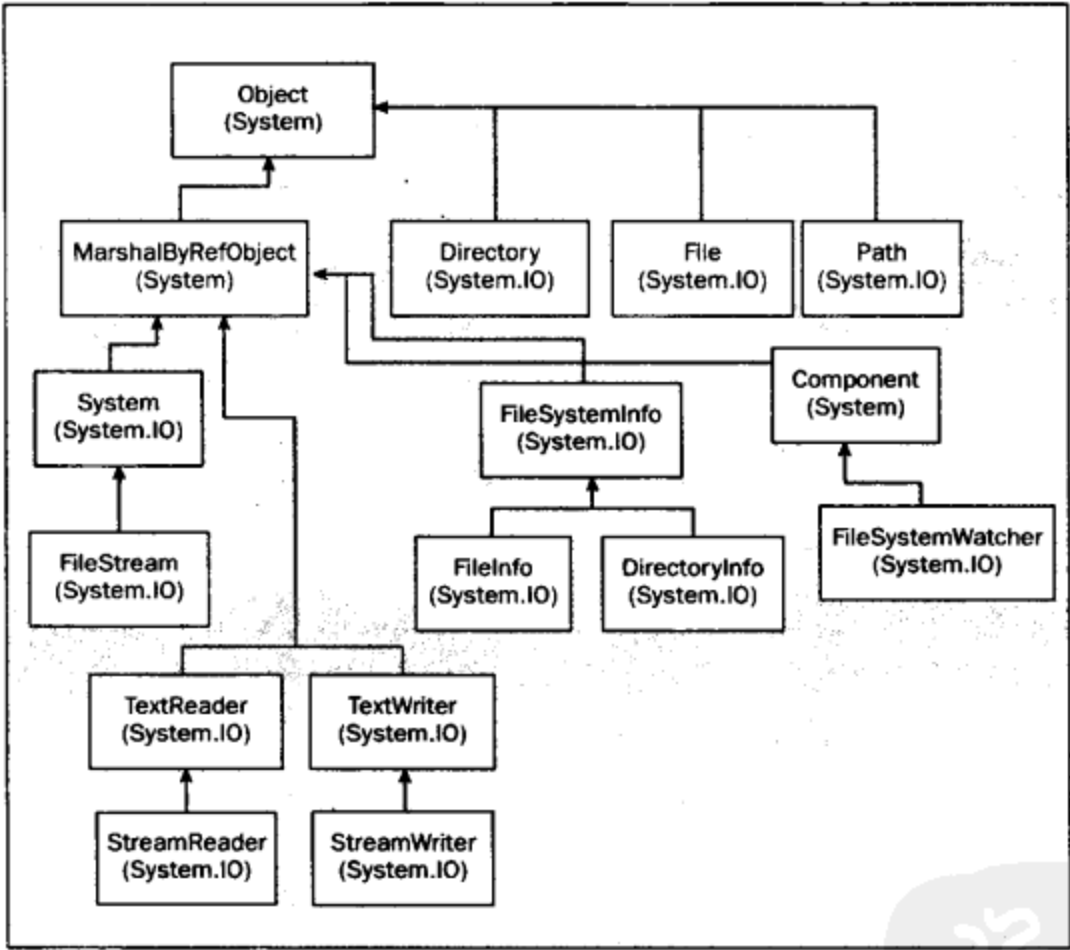


图 21-1

本章将介绍如表 21-1 中所示的一些类。

表 21-1

类	说 明
File	静态实用类，提供许多静态方法，用于移动、复制和删除文件
Directory	静态实用类，提供许多静态方法，用于移动、复制和删除目录

(续表)

类	说 明
Path	实用类，用于处理路径名称
FileInfo	表示磁盘上的物理文件，该类包含处理此文件的方法。要完成对文件的读写工作，就必须创建 Stream 对象
DirectoryInfo	表示磁盘上的物理目录，该类包含处理此目录的方法
FileStreamInfo	用作 FileInfo 和 DirectoryInfo 的基类，可以使用多态性同时处理文件和目录
FileStream	表示可写或可读，或二者均可的文件。此文件可以同步或异步地读写
StreamReader	从流中读取字符数据，可以使用 FileStream 将其创建为基类
StreamWriter	向流写入字符数据，可以使用 FileStream 将其创建为基类
FileSystemWatcher	FileSystemWatcher 是本章要介绍的最复杂的类。它用于监控文件和目录，提供了这些文件和目录发生变化时应用程序可以捕获的事件。在 Windows 编程技术中缺乏此功能，但是现在 .NET Framework 很容易对文件系统事件作出响应

本章还将介绍 System.IO.Compression 名称空间，它允许使用 GZIP 压缩或 Deflate 压缩模式读写压缩文件：

- DeflateStream——表示在写入时自动压缩数据或在读取时自动解压缩的流，使用 Deflate 算法来实现压缩。
- GZipStream——表示在写入时自动压缩数据或在读取时自动解压缩的流，压缩使用 GZIP 算法来实现。

最后，学习使用 System.Runtime.Serialization 名称空间及其子名称空间进行对象的序列化，主要介绍 System.Runtime.Serialization.Formatters.Binary 名称空间中的 BinaryFormatter 类，它允许把对象序列化为二进制数据流，并可以反序列化这些数据。

21.2.1 File 类和 Directory 类

File 和 Directory 实用类提供了许多静态方法，用于处理文件和目录。这些方法可以移动文件、查询和更新特性，创建 FileStream 对象。如第 8 章所述，可以在类上调用静态方法，而无需创建它们的实例。

File 类的一些最常用的静态方法如表 21-2 所示。

表 21-2

方 法	说 明
Copy()	将文件从源位置复制到目标位置
Create()	在指定的路径上创建文件
Delete()	删除文件
Open()	返回指定路径上的 FileStream 对象
Move()	将指定的文件移动到新位置。可以在新位置为文件指定不同的名称



Directory 类的一些常用的静态方法如表 21-3 所示。

表 21-3


方 法	说 明
CreateDirectory()	创建具有指定路径的目录
Delete()	删除指定的目录及其中的所有文件
GetDirectories()	返回表示指定目录下的目录名的 string 对象数组
EnumerateDirectories()	与 GetDirectories()类似，但返回目录名的 IEnumerable<string>集合
GetFiles()	返回在指定目录中的文件名的 string 对象数组
EnumerateFiles()	与 GetFiles()类似，但返回文件名的 IEnumerable<string>集合
GetFileSystemEntries()	返回在指定目录中的文件和目录名的 string 对象数组
Enumerate FileSystemEntries()	与 GetFileSystemEntries()类似，但返回文件和目录名的 IEnumerable<string>集合
Move()	将指定的目录移到新位置。可以在新位置为文件夹指定一个新名称

其中的 3 个 EnumerateXxx()方法是 .NET 4 新增的，在存在大量文件或目录时，其性能比对应的 GetXxx()方法好。

21.2.2 FileInfo 类

FileInfo 类不像 File 类，它不是静态的，没有静态方法，仅可用于实例化的对象。FileInfo 对象表示磁盘或网络位置上的文件。提供文件路径，就可以创建一个 FileInfo 对象：

```
FileInfo aFile = new FileInfo(@"C:\Log.txt");
```



本章处理的是表示文件路径的字符串，该字符串中有许多“\”字符，所以上述字符串的前缀@表示这个字符串应逐个字符地解释，“\”解释为“\”，而不解释为转义字符。如果没有@前缀，就需要使用“\\”代替“\”，以免把这个字符解释为转义字符。本章总是在字符串前面加上前缀@。

也可以把目录名传送给 FileInfo 的构造函数，但实际上这并不是很有效。这么做会用所有的目录信息初始化 FileInfo 的基类 FileSystemInfo，但 FileInfo 中与文件相关的专用方法或属性都不会工作。

FileInfo 类提供的许多方法类似于 File 类的方法，但是因为 File 是静态类，它需要一个字符串参数为每个方法调用指定文件位置。因此，下面的调用可以完成相同的工作：

```
FileInfo aFile = new FileInfo("Data.txt");

if (aFile.Exists)
    Console.WriteLine("File Exists");

if (File.Exists("Data.txt"))
    Console.WriteLine("File Exists");
```

这段代码检查文件 Data.txt 是否存在。注意，这里没有指定任何目录信息，这说明只检查当前的工作目录。这个目录包含调用此代码的应用程序。本章后面的“路径名和相对路径”一节会详细介绍这一内容。

大多数 FileInfo 方法采用这种方式反映 File 方法。在大多数情况下使用什么技术并不重要，但下面的规则有助于确定哪种技术更合适：

- 如果仅进行单一方法调用，则可以使用静态 File 类上的方法。在此，单一调用要快一些，因为 .NET Framework 不必实例化新对象，再调用方法。
- 如果应用程序在文件上执行几种操作，则实例化 FileInfo 对象并使用其方法就更好一些。这节省时间，因为对象已在文件系统上引用正确的文件，而静态类必须每次都寻找文件。

FileInfo 类也提供了与底层文件相关的属性，其中一些属性可以用来更新文件，其中很多属性都继承于 FileSystemInfo，所以可应用于 File 和 Directory 类。FileSystemInfo 类的属性如表 21-4 所示。

表 21-4

属 性	说 明
Attributes	使用 FileAttributes 枚举，获取或者设置当前文件或目录的特性
CreationTime, CreationTimeUtc	获取当前文件的创建日期和时间，可以使用 UTC 和非 UTC 版本
Extension	提取文件的扩展名。这个属性是只读的
Exists	确定文件是否存在，这是一个只读抽象属性，在 FileInfo 和 DirectoryInfo 中进行了重写
FullName	检索文件的完整路径，这个属性是只读的
LastAccessTime, LastAccessTimeUtc	获取或设置上次访问当前文件的日期和时间，可以使用 UTC 和非 UTC 版本
LastWriteTime, LastWriteTimeUtc	获取或设置上次写入当前文件的日期和时间，可以使用 UTC 和非 UTC 版本
Name	检索文件的完整路径，这是一个只读抽象属性，在 FileInfo 和 DirectoryInfo 中进行了重写

FileInfo 的专用属性如表 21-5 所示。

表 21-5

属 性	说 明
Directory	检索一个 DirectoryInfo 对象，表示包含当前文件的目录。这个属性是只读的
DirectoryName	返回文件目录的路径。这个属性是只读的
IsReadOnly	文件只读特性的快捷方式。这个属性也可以通过 Attributes 来访问
Length	获取文件的容量(以字节为单位)，返回 long 值。这个属性是只读的

FileInfo 对象本身不表示流。要读写文件，必须创建 Stream 对象。FileInfo 对象提供了几个返回实例化 Stream 对象的方法来帮助做到这一点。

21.2.3 DirectoryInfo 类

DirectoryInfo 类的作用类似于FileInfo 类。它是一个实例化的对象，表示计算机上的单一目录。同 FileInfo 类一样，在 Directory 和 DirectoryInfo 之间可以复制许多方法调用。选择使用 File 或 FileInfo 方法的规则也适用于 DirectoryInfo 方法：

- 如果进行单一调用，就使用静态 Directory 类。
- 如果进行一系列调用，则使用实例化的 DirectoryInfo 对象。

DirectoryInfo 类的大多数属性继承自 FileSystemInfo，与 FileInfo 类一样，但这些属性作用于目录上，而不是文件上。还有两个 DirectoryInfo 专用属性，如表 21-6 所示。

表 21-6

属 性	说 明
Parent	检索一个 DirectoryInfo 对象，表示包含当前目录的目录。这个属性是只读的
Root	检索一个 DirectoryInfo 对象，表示包含当前目录的根目录，例如 C:\目录。这个属性是只读的

21.2.4 路径名和相对路径

在.NET 代码中规定路径名时，可以使用绝对路径名，也可以使用相对路径名。绝对路径名显式地规定文件或目录来自于哪一个已知的位置，比如 C:驱动器。它的一个示例是 C:\Work\LogFile.txt。注意这个路径准确地定义了其位置。

相对路径名相对于一个起始位置。使用相对路径名时，无需规定驱动器或已知的位置；前面的当前工作目录就是起点，这是相对路径名的默认设置。例如，如果应用程序运行在 C:\Development\FileDemo 目录上，并使用相对路径 LogFile.txt，该文件就是 C:\Development\FileDemo\LogFile.txt。为了上移目录，要使用..字符串。这样，在同一个应用程序中，路径..\Log.txt 表示 C:\Development\Log.txt 文件。

如前所述，工作目录起初设置为运行应用程序的目录。当使用 VS 或 VCE 开发程序时，这就表示应用程序是所创建的项目文件夹下的几个目录。它通常位于 ProjectName\bin\Debug。要访问项目根文件夹中的文件，必须用..\..\上移两个目录，这在本章中很常见。

只要需要，就可以使用 Directory.GetCurrentDirectory()找出工作目录的当前设置，也可以使用 Directory.SetCurrentDirectory()设置新路径。

21.2.5 FileStream 对象

FileStream 对象表示在磁盘或网络路径上指向文件的流。这个类提供了在文件中读写字节的方法，但经常使用 StreamReader 或 StreamWriter 执行这些功能。这是因为 FileStream 类操作的是字节和字节数组，而 Stream 类操作的是字符数据。字符数据易于使用，但是有些操作，如随机文件访问(访问文件中间某点的数据)，就必须由 FileStream 对象执行，稍后对此进行介绍。

还有几种方法可以创建 FileStream 对象。构造函数具有许多不同的重载版本，最简单的构造函数仅有两个参数，即文件名和 FileMode 枚举值。

```
FileStream aFile = new FileStream(filename, FileMode.Member);
```

FileMode 枚举包含几个成员，指定了如何打开或创建文件。稍后介绍这些枚举成员。另一个常用的构造函数如下：

```
FileStream aFile = new FileStream(filename, FileMode.Member, FileAccess.Member);
```

第三个参数是 FileAccess 枚举的一个成员，它指定了流的作用。FileAccess 枚举的成员如表 21-7 所示。

表 21-7

成 员	说 明
Read	打开文件，用于只读
Write	打开文件，用于只写
ReadWrite	打开文件，用于读写

对文件进行非 FileAccess 枚举成员指定的操作会导致抛出异常。此属性的作用是，基于用户的身份验证级别改变用户对文件的访问权限。

在 FileStream 构造函数不使用 FileAccess 枚举参数的版本中，使用默认值 FileAccess.ReadWrite。

FileMode 枚举成员如表 21-8 所示。使用每个值会发生什么，取决于指定的文件名是否表示已有的文件。注意，这个表中的项表示创建流时该流指向文件中的位置，下一节将详细讨论这个主题。除非特别说明，否则流就指向文件的开头。

表 21-8

成 员	文 件 存 在	文 件 不 存 在
Append	打开文件，流指向文件的末尾，只能与枚举 FileAccess.Write 结合使用	创建一个新文件。只能与枚举 FileAccess.Write 结合使用
Create	删除该文件，然后创建新文件	创建新文件
CreateNew	抛出异常	创建新文件
Open	打开现有的文件，流指向文件开头	抛出异常
OpenOrCreate	打开文件，流指向文件开头	创建新文件
Truncate	打开现有文件，清除其内容。流指向文件开头，保留文件的初始创建日期	抛出异常

File 和 FileInfo 类都提供了 OpenRead()和 OpenWrite()方法，更易于创建 FileStream 对象。前者打开了只读访问的文件，后者只允许写入文件。这些都提供了快捷方式，因此不必以 FileStream 构造函数的参数形式提供前面所有的信息。例如，下面的代码行打开了用于只读访问的 Data.txt 文件：

```
FileStream aFile = File.OpenRead("Data.txt");
```

下面的代码执行同样的功能：

```
FileInfo aFileInfo = new FileInfo("Data.txt");
```

```
FileStream aFile = aFileInfo.OpenRead();
```

## 1. 文件位置

`FileStream`类维护内部文件指针，该指针指向文件中进行下一次读写操作的位置。在大多数情况下，当打开文件时，它就指向文件的开始位置，但是可以修改此指针。这允许应用程序在文件的任何位置读写，随机访问文件，或直接跳到文件的特定位置上。当处理大型文件时，这非常省时，因为马上可以找到正确的位置。

实现此功能的方法是 `Seek()`方法，它有两个参数：第一个参数规定文件指针移动距离(以字节为单位)。第二个参数规定开始计算的起始位置，用 `SeekOrigin` 枚举的一个值表示。`SeekOrigin` 枚举包含 3 个值：`Begin`、`Current` 和 `End`。

例如，下面的代码行将文件指针移动到文件的第 8 个字节，其起始位置就是文件的第 1 个字节：

```
aFile.Seek(8, SeekOrigin.Begin);
```

下面的代码行将指针从当前位置开始向前移动 2 个字节。如果在上面的代码行之后执行下面的代码，文件指针就指向文件的第 10 个字节：

```
aFile.Seek(2, SeekOrigin.Current);
```

注意读写文件时，文件指针会随之改变。在读取了 10 个字节之后，文件指针就指向被读取的第 10 个字节之后的字节。

也可以规定反向查找位置，这可以与 `SeekOrigin.End` 枚举值一起使用，查找靠近文件末端的位置。下面的代码会查找文件中倒数第 5 个字节：

```
aFile.Seek(-5, SeekOrigin.End);
```

采用这种方式访问的文件有时称为随机访问文件，因为应用程序可以访问文件中的任何位置。稍后介绍的 `Stream` 类可以连续地访问文件，但不允许以这种方式操作文件指针。



.NET 4 引入了一个新名称空间 `System.IO.MemoryMappedFiles`，它包含的类型(例如 `MemoryMappedFile`)提供了另一种随机访问特大型文件的方式。这个名称空间在本章不介绍，但如果需要访问特大型文件，就可以考虑使用这种方式。

## 2. 读取数据

使用 `FileStream` 类读取数据不像使用本章后面介绍的 `StreamReader` 类读取数据那样容易。这是因为 `FileStream` 类只能处理原始字节(raw byte)。处理原始字节的功能使 `FileStream` 类可以用于任何数据文件，而不仅仅是文本文件。通过读取字节数据，`FileStream` 对象可以用于读取诸如图像和声音的文件。这种灵活性的代价是，不能使用 `FileStream` 类将数据直接读入字符串，而使用 `StreamReader` 类却可以这样处理。但是有几种转换类可以很轻易地将字节数组转换为字符数组，或者将字符数组转换为字节数组。

`FileStream.Read()`方法是从 `FileStream` 对象所指向的文件中访问数据的主要手段。这个方法从文件中读取数据，再把数据写入一个字节数组。它有三个参数：第一个参数是传入的字节数组，用来接受 `FileStream` 对象中的数据。第二个参数是字节数组中开始写入数据的位置；它通常是 0，表示从数组开端向文件中写入数据。最后一个参数指定从文件中读出多少字节。

下面的示例演示了从随机访问文件中读取数据。要读取的文件实际是为此示例创建的类文件。

#### 试一试：从随机访问文件中读取数据

- (1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新的控制台应用程序 `ReadFile`。
- (2) 在 `Program.cs` 文件的顶部添加下面的 `using` 指令：



可从  
WROX.COM  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

代码段 `ReadFile\Program.cs`

- (3) 在 `Main()` 方法中添加下面的代码：

```
static void Main(string[] args)
{
    byte[] byData = new byte[200];
    char[] charData = new Char[200];

    try
    {
        FileStream aFile = new FileStream("../..//Program.cs", FileMode.Open);
        aFile.Seek(113, SeekOrigin.Begin);
        aFile.Read(byData, 0, 200);
    }
    catch (IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadKey();
        return;
    }

    Decoder d = Encoding.UTF8.GetDecoder();
    d.GetChars(byData, 0, byData.Length, charData, 0);

    Console.WriteLine(charData);
    Console.ReadKey();
}
```

- (4) 运行应用程序。结果如图 21-2 所示。



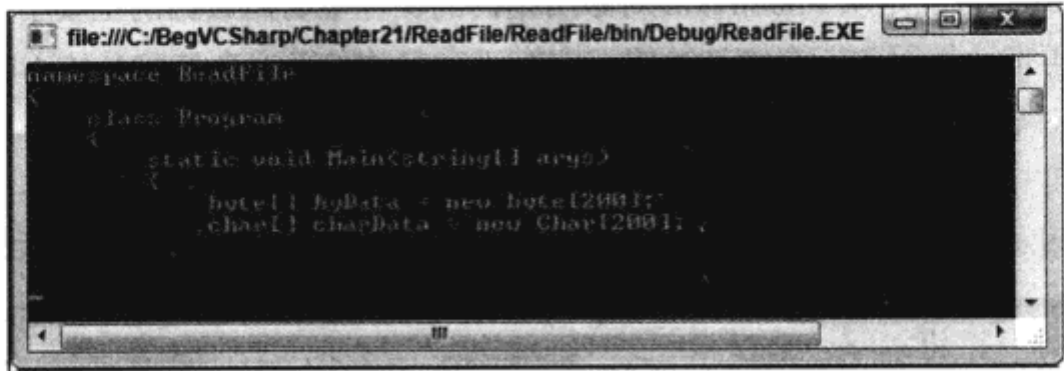


图 21-2

**示例的说明**

此应用程序打开自己的.cs 文件，用于从中读取。它在下面的代码行中使用.. 字符串向上逐级导航两个目录，找到该文件：

```
FileStream aFile = new FileStream("../../Program.cs", FileMode.Open);
```

下面两行代码执行查找工作，并从文件的具体位置读取字节：

```
aFile.Seek(113, SeekOrigin.Begin);
aFile.Read(byData, 0, 200);
```

第一行代码将文件指针移到文件的第 113 个字节。在 Program.cs 中，是指 namespace 中的“n”；其前面的 113 个字符是 using 指令。第二行将接下来的 200 个字节读入到 byData 字节数组中。注意，这两行代码封装在 try...catch 块中，以处理可能抛出的异常。

```
try
{
    aFile.Seek(113, SeekOrigin.Begin);
    aFile.Read(byData, 0, 100);
}
catch(IOException e)
{
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(e.ToString());
    Console.ReadKey();
    return;
}
```

文件 I/O 涉及到的所有操作都可以抛出 IOException 类型的异常。所有产品代码都必须包含错误处理，尤其是处理文件系统时更是如此。本章的所有示例都具有错误处理的基本形式。

从文件中获取了字节数组后，就需要将其转换为字符数组，以便在控制台显示它。为此，使用 System.Text 名称空间的 Decoder 类。此类用于将原始字节转换为更有用的项，比如字符：

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0);
```

这些代码基于 UTF-8 编码模式创建了 Decoder 对象。这就是 Unicode 编码模式。然后调用 GetChars()方法，此方法提取字节数组，将它转换为字符数组。完成之后，就可以将字符数组输出到控制台。



### 3. 写入数据

向随机访问文件中写入数据的过程与从中读取数据非常类似。首先需要创建一个字节数组；最简单的办法是首先构建要写入文件的字符数组。然后使用 `Encoder` 对象将其转换为字节数组，其用法非常类似于 `Decoder` 对象。最后调用 `Write()` 方法，将字节数组传送到文件中。

下面构建一个简单的示例演示其过程。

#### 试一试：将数据写入随机访问文件

- (1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新的控制台应用程序 `WriteFile`。
- (2) 在 `Program.cs` 文件顶部添加下面的 `using` 指令：



可从  
WTOX.COM  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

代码段 `WriteFile\Program.cs`

- (3) 在 `Main()` 方法中添加下面的代码：

```
static void Main(string[] args)
{
    byte[] byData;
    char[] charData;

    try
    {
        FileStream aFile = new FileStream("Temp.txt", FileMode.Create);
        charData = "My pink half of the drainpipe.".ToCharArray();
        byData = new byte[charData.Length];
        Encoder e = Encoding.UTF8.GetEncoder();
        e.GetBytes(charData, 0, charData.Length, byData, 0, true);

        // Move file pointer to beginning of file.
        aFile.Seek(0, SeekOrigin.Begin);
        aFile.Write(byData, 0, byData.Length);
    }
    catch (IOException ex)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(ex.ToString());
        Console.ReadKey();
        return;
    }
}
```

- (4) 运行该应用程序。稍后将其关闭。
- (5) 导航到应用程序目录——在目录中已经保存了文件，因为我们使用了相对路径。目录位于 `WriteFile\bin\Debug` 文件夹中。打开 `Temp.txt` 文件。可以在文件中看到如图 21-3 所示的文本。

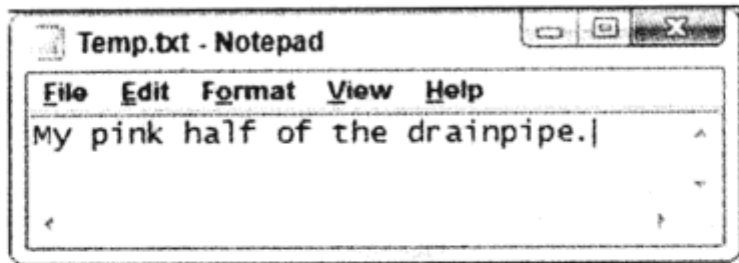


图 21-3

### 示例的说明

此应用程序在自己的目录中打开文件，并在文件中写入了一个简单的字符串。这个示例的结构非常类似于前面的示例，只是用 `Write()` 代替了 `Read()`，用 `Encoder` 代替了 `Decoder`。

下面的代码行使用 `String` 类的 `ToCharArray()` 静态方法，创建了字符数组。因为 C# 中的所有事物都是对象，文本 “My pink half of the drainpipe.” 实际上是一个 `String` 对象(尽管有点古怪)，所以甚至可以在字符串上调用这些静态方法。

```
CharData = " My pink half of the drainpipe. ".ToCharArray();
```

下面的代码行显示了如何将字符数组转换为 `FileStream` 对象需要的正确字节数组。

```
Encoder e = Encoding.UTF8.GetEncoder();  
e.GetBytes(charData, 0, charData.Length, byData, 0, true);
```

这次，要基于 UTF-8 编码方法来创建 `Encoder` 对象。也可以将 `Unicode` 用于解码。这里在写入流之前，需要将字符数据编码为正确的字节格式。在 `GetBytes()` 方法中可以完成这些工作，它可以将字符数组转换为字节数组，并将字符数组作为第一个参数(本例中的 `charData`)，将该数组中起始位置的下标作为第二个参数(0 表示数组的开头)。第三个参数是要转换的字符数量(`charData.Length`，`charData` 数组中的元素个数)。第四个参数是在其中放入数据的字节数组(`byData`)，第五个参数是在字节数组中开始写入位置的索引(0 表示 `byData` 数组的开头)。

最后一个参数决定在结束后 `Encoder` 对象是否应该更新其状态，即 `Encoder` 对象是否仍然保留它原来在字节数组中的内存位置。这有助于以后调用 `Encoder` 对象，但是当只进行单一调用时，这就没有什么意义。最后对 `Encoder` 的调用必须将此参数设置为 `true`，以清空其内存，释放对象，用于垃圾回收。

此后使用 `Write()` 方法向 `FileStream` 写入字节数组就变得非常简单：

```
aFile.Seek(0, SeekOrigin.Begin);  
aFile.Write(byData, 0, byData.Length);
```

与 `Read()` 方法一样，`Write()` 方法也有三个参数：要写入的数组，开始写入的数组索引和要写入的字节数。

### 21.2.6 StreamWriter 对象

操作字节数组比较麻烦，因为使用 `FileStream` 对象非常困难，那么，还有简单一些的方法吗？恐怕是没有了，因为有了 `FileStream` 对象，通常都会把它包装在 `StreamWriter` 或 `StreamReader` 中，并使用它们的方法来处理文件。如果不需要将文件指针改变到任意位置，这些类就很容易操作文件。

`StreamWriter` 类允许将字符和字符串写入到文件中，它处理底层的转换，向 `FileStream` 对象写入

数据。

还可以通过许多方法创建 `StreamWriter` 对象。如果已经有了 `FileStream` 对象，则可以使用此对象来创建 `StreamWriter` 对象：

```
FileStream aFile = new FileStream("Log.txt", FileMode.CreateNew);
StreamWriter sw = new StreamWriter(aFile);
```

也可以直接从文件中创建 `StreamWriter` 对象：

```
StreamWriter sw = new StreamWriter("Log.txt", true);
```

这个构造函数的参数是文件名和一个 `Boolean` 值，这个 `Boolean` 值规定是追加文件，还是创建新文件：

- 如果此值设置为 `false`，则创建一个新文件，或者截取现有文件并打开它。
- 如果此值设置为 `true`，则打开文件，保留原来的数据。如果找不到文件，则创建一个新文件。

与创建 `FileStream` 对象不同，创建 `StreamWriter` 对象不会提供一组类似的选项：除了使用 `Boolean` 值追加文件或创建新文件之外，根本没有像 `FileStream` 类那样指定 `FileMode` 属性的选项。而且，没有设置 `FileAccess` 属性的选项，因此总是拥有对文件的读/写权限。为了使用高级参数，必须先在 `FileStream` 构造函数中指定这些参数，然后在 `FileStream` 对象中创建 `StreamWriter`，如下面的示例所示。

#### 试一试：把数据写入输出流

(1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新的控制台应用程序 `StreamWrite`。

(2) 再次使用 `System.IO` 名称空间，因此在 `Program.cs` 文件靠近顶部的位置添加下面的 `using` 指令：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

代码段 `StreamWrite\Program.cs`

(3) 在 `Main()` 方法中添加下面的代码：

```
static void Main(string[] args)
{
    try
    {
        FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate);
        StreamWriter sw = new StreamWriter(aFile);

        bool truth = true;
        // Write data to file.
        sw.WriteLine("Hello to you.");
        sw.WriteLine("It is now {0} and things are looking good.",
            DateTime.Now.ToLongDateString());
    }
}
```

```

        sw.Write("More than that,");
        sw.Write(" it's {0} that C# is fun.", truth);
        sw.Close();
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }
}

```

(4) 生成并运行该项目。如果找不到错误，则项目会很快运行，并关闭。因为我们在控制台上没有显示任何内容，所以在控制台中无法看到程序的执行情况。

(5) 进入应用程序目录，找到 Log.txt 文件，它位于 StreamWriter\bin\Debug 文件夹，这是因为我们使用了相对路径。

(6) 打开文件，可以看到图 21-4 所示的文本。

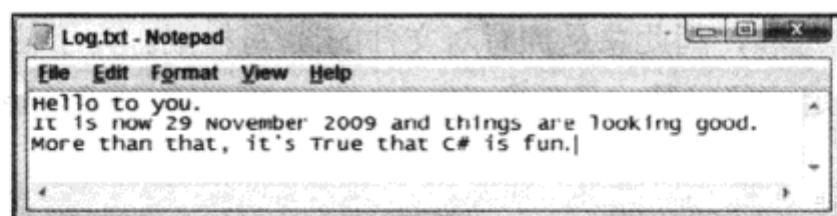


图 21-4

#### 示例的说明

这个简单的应用程序演示了 StreamWriter 类的两个最重要的方法：Write()和 WriteLine()。这两个方法具有许多重载的版本，可以完成更高级的文件输出，但是本示例只使用基本的字符串输出。

WriteLine()方法会写入传递给它的字符串，其后跟有换行符。在此示例中，下一个写入操作在新行上开始。

如同可以向控制台写入格式化数据一样，也可以向文件写入格式化数据。例如，可以使用标准格式化参数把变量的值写入文件：

```

sw.WriteLine("It is now {0} and things are looking good.",
    DateTime.Now.ToLongDateString());

```

DateTime.Now 存储当前日期，ToLongDateString()方法用于把这个日期转换为易于读取的格式。

Write()方法只是把传送给它的字符串写入文件，但不追加换行符，因此可以使用多个 Write()语句写入完整的句子或段落。

```

sw.Write("More than that,");
sw.Write(" it's {0} that C# is fun.", truth);

```

这里也使用了格式化参数，这次是使用 Write()显示布尔值 truth。前面把这个变量设置为 true，其值会自动格式化，转换为字符串“True”。

可以使用 Write()和格式化参数写入用逗号分隔的文件：

```

[StreamWriter object].Write("{0},{1},{2}", 100, "A nice product", 10.50);

```

在更复杂的示例中，这些数据还可以来自数据库或其他数据源。

### 21.2.7 StreamReader 对象

输入流用于从外部源中读取数据。很多情况下，数据源是磁盘上的文件或网络的某些位置。任何可以发送数据的位置都可以是数据源，比如网络应用程序、Web 服务甚至是控制台。

用来从文件中读取数据的类是 `StreamReader`。同 `StreamWriter` 一样，这是一个通用类，可以用于任何流。下面的示例会再次围绕 `FileStream` 对象构造 `StreamReader` 类，使它指向正确的文件。

`StreamReader` 对象的创建方式与 `StreamWriter` 对象非常类似。创建它的最常见方式是使用前面创建的 `FileStream` 对象：

```
FileStream aFile = new FileStream("Log.txt", FileMode.Open);
StreamReader sr = new StreamReader(aFile);
```

同 `StreamWriter` 一样，`StreamReader` 类可以直接在包含具体文件路径的字符串中创建：

```
StreamReader sr = new StreamReader("Log.txt");
```

#### 试一试：从输入流中读取数据

- (1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新控制台应用程序 `StreamRead`。
- (2) 必须再次输入 `System.IO` 名称空间，因此将下面的代码放在 `Program.cs` 文件的靠近顶部的位置：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

代码段 `StreamRead\Program.cs`

- (3) 在 `Main()` 方法中添加下面的代码：

```
static void Main(string[] args)
{
    string strLine;

    try
    {
        FileStream aFile = new FileStream("Log.txt", FileMode.Open);
        StreamReader sr = new StreamReader(aFile);
        Line = sr.ReadLine();
        // Read data in line by line.
        while(strLine != null)
        {
            Console.WriteLine(strLine);
            Line = sr.ReadLine();
        }
        sr.Close();
    }
}
```

```

catch(IOException e)
{
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(e.ToString());
    return;
}
Console.ReadKey();
}

```

(4) 把前面示例中创建的Log.txt 文件复制到 StreamReader\bin\Debug 目录下。如果没有 Log.txt 文件, FileStream 构造函数找不到该文件, 就会抛出异常。

(5) 运行该应用程序, 可以看到写入到控制台的文件文本, 如图 21-5 所示。

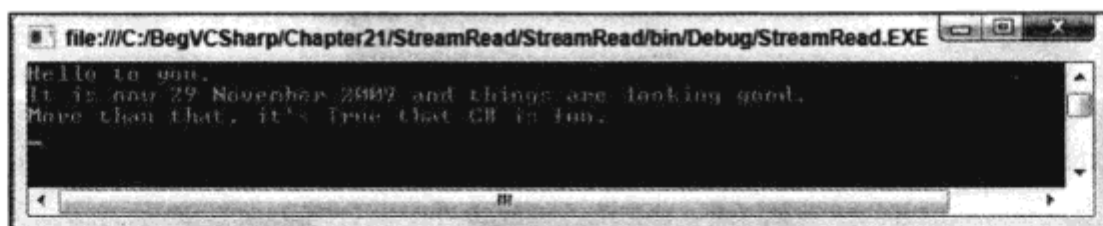


图 21-5

### 示例的说明

这个应用程序与前面的应用程序非常类似。其明显的区别就是, 它是在读取数据, 而不是写入数据。同前面一样, 只有导入 System.IO 名称空间, 才能访问需要的类。

使用 ReadLine()方法从文件中读取文本。这个方法读取换行符之前的文本, 并以字符串的形式返回结果文本。当到达文件尾时, 该方法就返回空值, 通过这种方法可以测试文件是否已到达了尾部。注意使用了 while 循环, 以便确保在执行循环体的代码之前读取的行不为空, 这样就只显示文件的有效内容:

```

strLine = sr.ReadLine();
while(strLine != null)
{
    Console.WriteLine(strLine);
    strLine = sr.ReadLine();
}

```

### 1. 读取数据

ReadLine()方法不是在文件中访问数据的唯一方法。StreamReader 类还包含许多读取数据的方法。

读取数据最简单的方法是 Read()。此方法将流的下一个字符作为正整数值返回, 如果到达了流的结尾处, 则返回 -1。使用 Convert 实用类可以把这个值转换为字符。在上面的示例中, 程序的主体可以按如下方式重新编写:

```

StreamReader sr = new StreamReader(aFile);
int nChar;
nChar = sr.Read();
while(nChar != -1)
{
    Console.Write(Convert.ToChar(nChar));
    nChar = sr.Read();
}
sr.Close();

```



对于小型文件，可以使用一个非常方便的方法 `ReadToEnd()`。此方法读取整个文件，并将其作为字符串返回。在此，前面的应用程序可以简化为：

```
StreamReader sr = new StreamReader(aFile);
Line = sr.ReadToEnd();
Console.WriteLine(strLine);
sr.Close();
```

这似乎非常容易和方便，但必须小心。将所有数据读取到字符串对象中，会迫使文件中的数据放到内存中。应根据数据文件的大小禁止这样处理。如果数据文件非常大，最好将数据留在文件中，并使用 `StreamReader` 的方法访问文件。

处理大型文件的另一种方式是 .NET 4 中新增的静态方法 `File.ReadLines()`。实际上，`File` 的几个静态方法可以用于简化文件数据的读写，但这个方法特别有趣，因为它返回 `IEnumerable<string>` 集合。可以迭代这个集合中的字符串，一次读取文件中的一行。使用这个方法，可以把前面的示例重写为：

```
foreach (string alternativeLine in File.ReadLines("Log.txt"))
    Console.WriteLine(alternativeLine);
```

可以看出，在 .NET 中，可以通过多种不同的方式获得相同的结果——读取文件中的数据。您可以选择其中最适当的技术。

## 2. 用分隔符分隔的文件

用分隔符分隔的文件是一种常见的数据存储形式，由许多旧系统使用，如果应用程序必须与这种系统协作，就会经常遇到用分隔符分隔的数据格式。最常见的分隔符是逗号，例如 Excel 电子表格、Access 数据库或 SQL Server 数据库中的数据都可以导出为用逗号分隔的值(CSV)文件。

前面介绍了如何使用 `StreamWriter` 类编写使用这种方法存储数据的文件。使用逗号分隔的文件也易于读取。如第 5 章所述，`String` 类的 `Split()` 方法可以用于将字符串转换为基于所提供的分隔符的数组。如果规定逗号为分隔符，它就会创建尺度合理的字符串数组，其中包含在初始逗号分隔的字符串中的全部数据。

下面的示例将说明其用途。这个示例处理用逗号分隔的值，把它们加载到 `List<Dictionary<string, string>>` 对象中。这个示例非常通用，如果需要处理用逗号分隔的值，就可以在自己的应用程序中使用这项技术。

### 试一试：逗号分隔的值

(1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新控制台应用程序 `CommaValues`。

(2) 将下面的代码放在 `Program.cs` 文件的顶部。只有导入 `System.IO` 名称空间，才能进行文件处理：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

代码段 `CommaValues\Program.cs`



(3) 在 Program.cs 主体的 Main()方法之前, 添加下面的 GetData()方法:

```
private static List<Dictionary<string, string>> GetData(
    out List<string> columns)
{
    string line;
    string[] stringArray;
    char[] charArray = new char[] {','};
    List<Dictionary<string, string>> data =
        new List<Dictionary<string, string>>();
    columns = new List<string>();
    try
    {
        FileStream aFile = new FileStream(@"..\..\SomeData.txt", FileMode.Open);
        StreamReader sr = new StreamReader(aFile);

        // Obtain the columns from the first line.
        // Split row of data into string array
        line = sr.ReadLine();
        stringArray = line.Split(charArray);

        for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
        {
            columns.Add(stringArray[x]);
        }

        line = sr.ReadLine();
        while (line != null)
        {
            // Split row of data into string array
            stringArray = line.Split(charArray);
            Dictionary<string, string> dataRow = new Dictionary<string, string>();

            for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
            {
                dataRow.Add(columns[x], stringArray[x]);
            }

            data.Add(dataRow);

            line = sr.ReadLine();
        }

        sr.Close();
        return data;
    }
    catch (IOException ex)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(ex.ToString());
        Console.ReadLine();
        return data;
    }
}
```

(4) 在 Main()方法中添加下面的代码:

```
static void Main(string[] args)
{
    List<string> columns;
    List<Dictionary<string, string>> myData = GetData(out columns);

    foreach (string column in columns)
    {
        Console.Write("{0,-20}", column);
    }
    Console.WriteLine();

    foreach (Dictionary<string, string> row in myData)
    {
        foreach (string column in columns)
        {
            Console.Write("{0,-20}", row[column]);
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}
```

(5) 从 Project | Add New Item 对话框中选择 Text File, 添加一个新文本文件 SomeData.txt。

(6) 在新文本文件中输入下面的文本:



可从  
wrox.com  
下载源代码

```
ProductID,Name,Price
1,Spiky Pung,1000
2,Gloop Galloop Soup,25
4,Hat Sauce,12
```

代码段 CommaValues\SomeData.txt

(7) 运行该应用程序——可以看到写入到控制台的文件文本, 如图 21-6 所示。

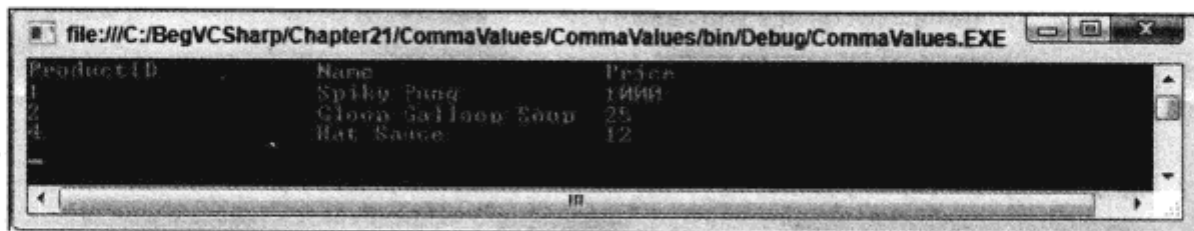


图 21-6

#### 示例的说明

与前面的示例相同, 这个应用程序也逐行地把文件的数据读入字符串。但是, 因为这是一个包含用逗号分隔的文本值的文件, 所以按不同的方式处理它。而且, 还把读取的值存储在一些数据结构中。

首先需要查看一些用逗号分隔的数据:

```
ProductID,Name,Price
1,Spiky Pung,1000
```

注意第一行包含数据列的名称，其他行包含数据。所以，过程应从文件的第一行中获取列名，再从其他的行中获取数据。

`GetData()`方法声明为 `static`，所以可以在不创建类实例的情况下调用它。这个方法返回一个 `List<Dictionary<string, string>>` 对象，下面要创建这个对象，并使用逗号分隔的文本文件中的数据填充它。该方法还返回一个包含标题名的 `List<string>` 对象。下面的代码初始化了这些对象：

```
List<Dictionary<string, string>> data = new List<Dictionary<string, string>>();
columns = new List<string>();
```

`columns` 包含逗号分隔的文本文件中第一行的列名，`data` 包含后续各行的值。

首先创建一个 `FileStream` 对象，再构建一个 `StreamReader` 对象，就像前面的示例那样。现在就可以读取文件的第一行，从这个字符串中创建一个字符串数组：

```
line = sr.ReadLine();
stringArray = line.Split(charArray);
```

第5章介绍了 `Split()` 方法，它的参数是一个字符数组，在这里该数组只包含“，”，所以 `stringArray` 包含的字符串数组是在 `line` 的每个“，”实例处进行分隔而构建的。当前是从文件的第一行中读取数据，这一行包含数据列的名称，所以需要迭代 `stringArray` 中的每个字符串，把它们添加到 `columns` 中：

```
for (int x = 0; x <= strArray.GetUpperBound(0); x++)
{
    columns.Add(strArray[x]);
}
```

数据有了列名后，就可以读取数据了。其代码与前面的 `StreamRead` 示例的代码相同，只是需要把 `Dictionary<string, string>` 对象添加到 `data` 中：

```
line = sr.ReadLine();
while (line != null)
{
    // Split row of data into string array.
    stringArray = line.Split(charArray);
    Dictionary<string, string> dataRow = new Dictionary<string, string>();

    for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
    {
        dataRow.Add(columns[x], stringArray[x]);
    }

    data.Add(dataRow);

    line = sr.ReadLine();
}
```

对于文件中的每一行，都创建一个新的 `Dictionary<string, string>` 对象，用一行数据填充它。这个集合中的每一项都有一个对应于列名的键和一个值，这个值对应于该行的列值。键从前面创建的 `columns` 对象中提取，值从使用 `Split()` 获得的字符串中提取，`Split()` 方法用于从数据文件中提取文本行。

读取完文件中的所有数据，就关闭 `StreamReader`，然后返回数据。`Main()`方法中的代码把从 `GetData()`方法获得的数据放在变量 `myData` 和 `columns` 中，并在控制台上显示这些信息。首先显示每一列的名称：

```
foreach (string column in columns)
{
    Console.Write("{0,-20}", column);
}
Console.WriteLine();
```

格式化字符串 `{0,-20}` 中的 `-20` 部分确保显示的名称在 20 字符长的列中左对齐，这有助于格式化显示结果。

最后迭代 `myData` 集合中的每个 `Dictionary<string, string>` 对象，显示该行中的值，这里再次使用格式化字符串来格式化输出结果。

```
foreach (Dictionary<string, string> row in myData)
{
    foreach (string column in columns)
    {
        Console.Write("{0,-20}", row[column]);
    }
    Console.WriteLine();
}
```

可以看到，使用 .NET Framework 从(CSV)文件中提取有意义的数据非常容易。这种技术也很容易与后续章节中介绍的数据访问技术合并使用，即 CSV 文件中的数据可以像其他数据源(如数据库)中的数据那样操作。但从 CSV 文件中提取的数据没有数据类型信息。当前是把所有的数据看作字符串，但对于企业级的商务应用程序来说，就需要更进一步，给提取的数据添加类型信息。这可能来自存储在 CSV 文件中的附加信息，可以手工进行配置，也可以从文件的字符串中推断，具体取决于特定的应用程序。

第 22 章介绍的 XML 也是一种存储和传输数据的优秀方法，而 CSV 文件仍然非常普遍，还会使用相当长的时间。逗号分隔文件的优点是非常简洁，因此要比 XML 文件小些。

### 21.2.8 读写压缩文件

在处理文件时，常常会发现文件中有许多空格，耗尽了硬盘空间。图形和声音文件尤其如此。读者可能使用过能压缩文件和解压文件的工具，当希望带着文件到其他地方去或者把文件邮寄给朋友时，使用这些工具是很方便的。`System.IO.Compression` 名称空间就包含能在代码中压缩文件的类，这些类使用 GZIP 或 Deflate 算法，这两种算法都是公开的、免费的，任何人都可以使用。

但压缩文件并不只是把它们压缩一下就完事了。商业应用程序允许把多个文件放在一个压缩文件中，本节介绍的内容要简单得多：只是把文本数据保存在压缩文件中。不能在外部实用程序中访问这个文件，但这个文件比未压缩版本要小得多。

`System.IO.Compression` 名称空间中有两个压缩流类 `DeflateStream` 和 `GZipStream`，它们的工作方式非常类似。对于这两个类，都要用已有的流初始化它们，对于文件，流就是 `FileStream` 对象。此后就可以把它们用于 `StreamReader` 和 `StreamWriter` 了，就像使用其他流一样。除此之外，只需指定流是用于压缩(保存文件)还是解压缩(加载文件)，类就知道要对传送给它的数据执行什么操作。这最

好用一个示例来加以说明。

### 试一试：读写压缩数据

(1) 在 C:\BegVCSharp\Chapter21 目录中创建一个新的控制台应用程序 Compressor。

(2) 把下面的代码放在 Program.cs 靠近顶部的位置。需要导入 System.IO 名称空间才能处理文件，导入 System.IO.Compression 才能使用压缩类：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.IO.Compression;
```

代码段 Compressor\Program.cs

(3) 把下面的方法添加到 Program.cs 中 Main()方法的前面：

```
static void SaveCompressedFile(string filename, string data)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Create, FileAccess.Write);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Compress);
    StreamWriter writer = new StreamWriter(compressionStream);
    writer.Write(data);
    writer.Close();
}

static string LoadCompressedFile(string filename)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Open, FileAccess.Read);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Decompress);
    StreamReader reader = new StreamReader(compressionStream);
    string data = reader.ReadToEnd();
    reader.Close();
    return data;
}
```

(4) 给 Main()方法添加如下代码：

```
static void Main(string[] args)
{
    try
    {
        string filename = "compressedFile.txt";

        Console.WriteLine(
            "Enter a string to compress (will be repeated 1000 times):");
        string sourceString = Console.ReadLine();
        StringBuilder sourceStringMultiplier =
```

```

        new StringBuilder(sourceString.Length * 100);
    for (int i = 0; i < 100; i++)
    {
        sourceStringMultiplier.Append(sourceString);
    }
    sourceString = sourceStringMultiplier.ToString();
    Console.WriteLine("Source data is {0} bytes long.", sourceString.Length);

    SaveCompressedFile(filename, sourceString);
    Console.WriteLine("\nData saved to {0}.", filename);

    FileInfo compressedFileData = new FileInfo(filename);
    Console.WriteLine("Compressed file is {0} bytes long.",
        compressedFileData.Length);

    string recoveredString = LoadCompressedFile(filename);
    recoveredString = recoveredString.Substring(0, recoveredString.Length / 100);
    Console.WriteLine("\nRecovered data: {0}", recoveredString);

    Console.ReadKey();
}
catch (IOException ex)
{
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(ex.ToString());
    Console.ReadKey();
}
}

```

(5) 运行应用程序，输入一个长度合理的长字符串，结果如图 21-7 所示。

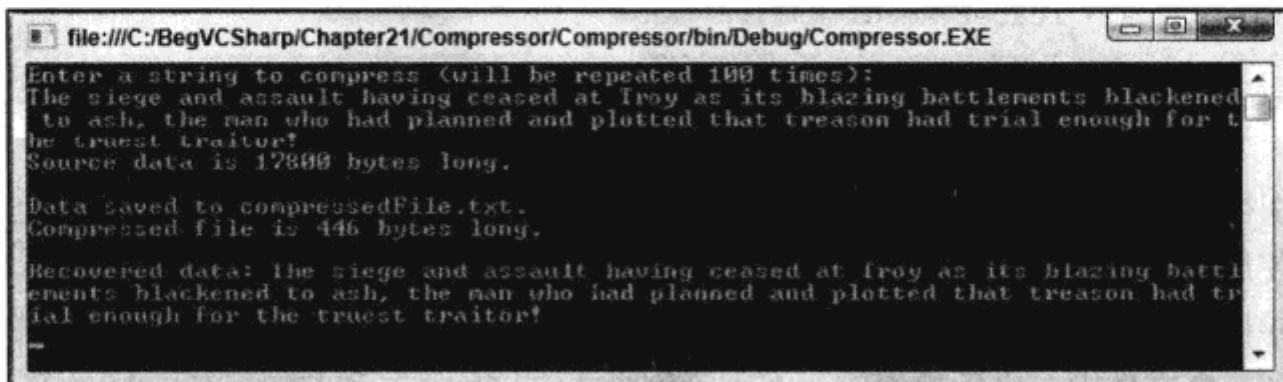


图 21-7

(6) 在记事本中打开 compressedFile.txt，文本如图 21-8 所示。

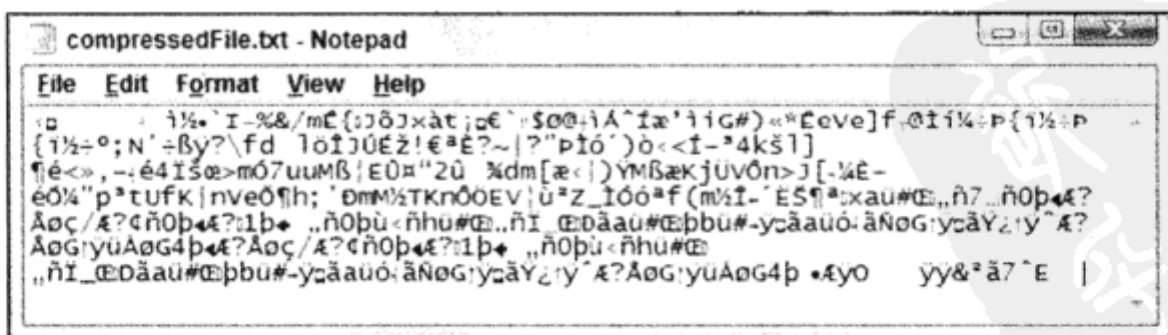


图 21-8

### 示例的说明

在这个示例中，定义了两个方法，用于保存和加载已压缩的文本文件。第一个方法是 `SaveCompressedFile()`，如下所示：

```
static void SaveCompressedFile(string filename, string data)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Create, FileAccess.Write);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Compress);
    StreamWriter writer = new StreamWriter(compressionStream);
    writer.Write(data);
    writer.Close();
}
```

代码首先创建一个 `FileStream` 对象，然后使用它创建一个 `GZipStream` 对象。注意，可以用 `DeflateStream` 替换这段代码中的所有 `GZipStream`——这两个类的工作方式相同。使用 `CompressionMode.Compress` 枚举值指定数据要进行压缩，然后使用 `StreamWriter` 把数据写入文件。

`LoadCompressedFile()` 方法与 `SaveCompressedFile()` 方法正好相对，它不是保存到文件名中，而是把压缩的文件加载到字符串中：

```
static string LoadCompressedFile(string filename)
{
    FileStream fileStream =
        new FileStream(filename, FileMode.Open, FileAccess.Read);
    GZipStream compressionStream =
        new GZipStream(fileStream, CompressionMode.Decompress);
    StreamReader reader = new StreamReader(compressionStream);
    string data = reader.ReadToEnd();
    reader.Close();
    return data;
}
```

其区别很显然：使用了不同的 `FileMode`、`FileAccess` 和 `CompressionMode` 枚举值来加载和解压数据，使用 `StreamReader` 从文件中提取出未压缩的文本。

`Main()` 中的代码是这些方法的一个简单测试。它请求一个字符串，把字符串复制 100 次，再把它压缩到一个文件中，之后检索它。在本示例中，Sir Gawain 和 Green Knight 的开头部分重复 100 次就有 17 800 个字符，但压缩后只占用 446 个字节，压缩率是 40:1。应该承认，这有以偏盖全之嫌，GZIP 算法很适合重复数据，但这里只是演示压缩过程而已。

我们还查看了存储在压缩文件中的文本。显然，它是不可读的。这说明可以在应用程序之间共享数据。因为是用已知的算法压缩文件，应用程序肯定知道如何解压缩。

## 21.3 序列化对象

应用程序常常需要在硬盘上存储数据。本章前面介绍了逐段构建文本和数据文件，但这常常不是最简便的方式。有时最好以对象形式存储数据。

.NET Framework 在 `System.Runtime.Serialization` 和 `System.Runtime.Serialization.Formatters` 名称



空间中提供了序列化对象的基础架构，这两个名称空间中的一些类实现了这个基础架构。Framework 中有两个可用的实现方式：

- `System.Runtime.Serialization.Formatters.Binary`：这个名称空间包含了 `BinaryFormatter` 类，它能把对象序列化为二进制数据，把二进制数据序列化为对象。
- `System.Runtime.Serialization.Formatters.Soap`：这个名称空间包含了 `SoapFormatter` 类，它能把对象序列化为 SOAP 格式的 XML 数据，把 SOAP 格式的 XML 数据序列化为对象。

本章只介绍 `BinaryFormatter`，因为还没有学习 XML 数据。实际上不鼓励使用 `SoapFormatter` 格式化器，但希望进行可读的序列化时，仍可以使用它。这两个类都实现了 `IFormatter` 接口，这里讨论的很多内容适用于这两个类。



.NET Framework 中的另外两个类也实现了 `IFormatter` 接口，第一个是 `ObjectStateFormatter`，用于在 ASP.NET 中序列化 viewstate，另一个是 `NetDataContractSerializer`，用于序列化 WCF 数据合同。

`IFormatter` 接口提供了如表 21-9 中所示的方法。

表 21-9	
方 法	说 明
<code>void Serialize(Stream stream, object source)</code>	把 source 序列化为 stream
<code>object Deserialize(Stream stream)</code>	反序列化 stream 中的数据，返回得到的对象

重要的是，为了便于本章的讨论，这些方法都处理流，以便把这些方法与本章前面介绍的文件访问技术联系起来——可以使用 `FileStream` 对象。

所以，使用 `BinaryFormatter` 进行序列化非常简单：

```
IFormatter serializer = new BinaryFormatter();
serializer.Serialize(myStream, myObject);
```


反序列化同样也很简单：

```
IFormatter serializer = new BinaryFormatter();
MyObjectType myNewObject = serializer.Deserialize(myStream) as MyObjectType;
```

显然，需要流和对象才能进行处理，但上面的代码对于几乎所有的情况都是正确的。下面的示例将把这些代码用于实际。

试一试：对象的序列化和反序列化

- (1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新的控制台应用程序 `ObjectStore`。
- (2) 给项目添加一个新类 `Product`，修改代码，如下所示：



可从  
wrox.com  
下载源代码

```
namespace ObjectStore
{
    public class Product
```

```

{
    public long Id;
    public string Name;
    public double Price;

    [NonSerialized]
    string Notes;

    public Product(long id, string name, double price, string notes)
    {
        Id = id;
        Name = name;
        Price = price;
        Notes = notes;
    }

    public override string ToString()
    {
        return string.Format("{0}: {1} (${2:F2}) {3}", Id, Name, Price, Notes);
    }
}
}

```

---

代码段 ObjectStore\Product.cs

---

(3) 把下面的代码放在 Program.cs 的顶部。需要导入 System.IO 名称空间, 才能处理文件, 导入其他名称空间, 才能进行序列化:



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

```

---

代码段 ObjectStore\Program.cs

---

(4) 把下面的代码添加到 Program.cs 的 Main()方法中:

```

static void Main(string[] args)
{
    try
    {
        // Create products.
        List<Product> products = new List<Product>();
        products.Add(new Product(1, "Spiky Pung", 1000.0, "Good stuff.));
        products.Add(new Product(2, "Gloop Galloop Soup", 25.0, "Tasty.));
        products.Add(new Product(4, "Hat Sauce", 12.0, "One for the kids.));

        Console.WriteLine("Products to save:");
        foreach (Product product in products)
        {
            Console.WriteLine(product);
        }
    }
}

```

```

    }
    Console.WriteLine();

    // Get serializer.
    IFormatter serializer = new BinaryFormatter();

    // Serialize products.
    FileStream saveFile =
        new FileStream("Products.bin", FileMode.Create, FileAccess.Write);
    serializer.Serialize(saveFile, products);
    saveFile.Close();

    // Deserialize products.
    FileStream loadFile =
        new FileStream("Products.bin", FileMode.Open, FileAccess.Read);
    List<Product> savedProducts =
        serializer.Deserialize(loadFile) as List<Product>;
    loadFile.Close();

    Console.WriteLine("Products loaded:");
    foreach (Product product in savedProducts)
    {
        Console.WriteLine(product);
    }
}
catch (SerializationException e)
{
    Console.WriteLine("A serialization exception has been thrown!");
    Console.WriteLine(e.Message);
}
catch (IOException e)
{
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(e.ToString());
}

Console.ReadKey();
}

```

(5) 运行应用程序，结果如图 21-9 所示。

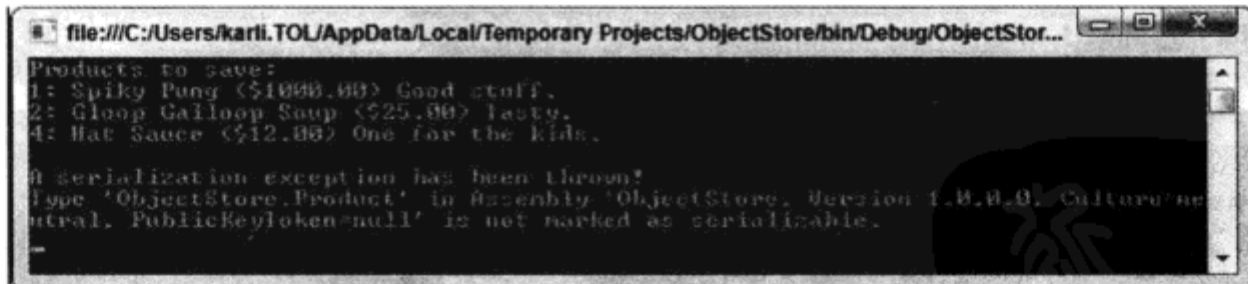


图 21-9

(6) 修改 Product.cs 中的代码，如下所示：

```

namespace ObjectStore
{
    [Serializable]
    public class Product

```



可从  
wrox.com  
下载源代码

```
{
    ...
}
```

代码段 ObjectStore\Product.cs

(7) 再次运行应用程序，结果如图 21-10 所示。

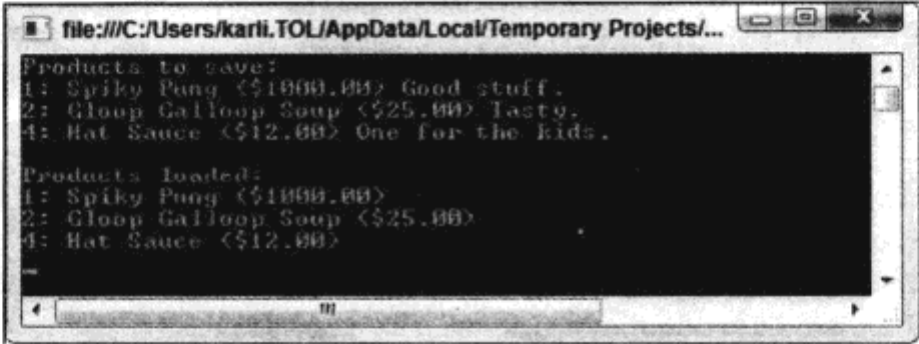


图 21-10

(8) 在记事本中打开 Products.bin，文本如图 21-11 所示。

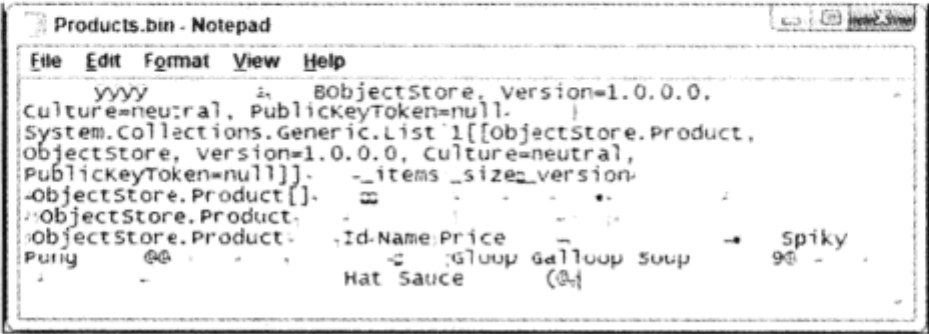


图 21-11

示例的说明

本示例创建了一个 Product 对象集合，把集合保存到磁盘上，然后重新加载它。但第一次运行应用程序时，抛出了一个异常，因为 Product 对象没有标记为“可序列化”。

.NET Framework 要求把对象标记为可序列化，才能序列化它们。这有许多原因，包括：

- 一些对象序列化的效果不佳。例如，它们需要引用只有它们本身位于内存中时才存在的本地数据。
- 一些对象包含敏感的数据，这些数据不应以不安全的方式保存或传送到另一个进程中。

如本例所示，使用 Serializable 特性就可以把对象标记为可序列化：

```
namespace ObjectStore
{
    [Serializable]
    public class Product
    {
        ...
    }
}
```

这里要注意，这个特性并没有由派生类继承，它必须应用于要进行序列化的每个类。另外，用于生成 Product 对象集合的 List<T>类有这个特性，否则，把它应用于 Product 就无益于使集合可序列化。

products 集合成功序列化和反序列化(在第二次尝试中)时，要注意另一个重要的问题。只重新构

建了 Id、Name 和 Price 字段，这是因为使用了另一个特性 NonSerialized。

```
[NonSerialized]
string Notes;
```

任何成员都可以用这个特性标记，但不能和其他成员一起存储。如果只有一个字段或属性包含敏感数据，这就是很有用的。

示例中还介绍了最后保存的数据。注意这里的一些数据是人们可以看懂的，这可能与我们的期望的不同。BinaryFormatter 类并没有试图把数据隐藏起来。当然，这里使用了流，在把它保存到磁盘上，或者加载并应用自己的加密算法时，很容易截取其中的数据。压缩也是这样——使用上一节中的技术，很容易在把对象数据保存到磁盘上时压缩它们。

序列化涉及许多内容，但前面介绍的基础知识已经足够了。下面要研究的一个较高级技术是使用 ISerializable 接口的定制序列化，它允许定制序列化的数据。例如，在发布后升级类时，这是很重要的。修改可序列化的成员，会使所保存的已有数据不再可读，除非提供自己的逻辑来保存和获取数据。

### 21.4 监控文件系统

有时，应用程序所需要完成的工作不仅仅限于从文件系统中读写文件。例如，知道修改文件或目录的时间非常重要。.NET Framework 允许方便地创建完成这些任务的定制应用程序。

帮助完成这些任务的类是 FileSystemWatcher。这个类提供了几个应用程序可以捕获的事件。应用程序可以对文件系统事件作出响应。

使用 FileSystemWatcher 的基本过程非常简单。首先必须设置一些属性，指定监控的位置、内容以及引发应用程序要处理的事件的时间。然后给 FileSystemWatcher 提供定制事件处理程序的地址，当发生重要事件时，FileSystemWatcher 就可以进行调用。最后打开 FileSystemWatcher，等待事件。

在启用 FileSystemWatcher 对象之前必须设置的属性如表 21-10 所示。

表 21-10

属 性	说 明
Path	设置要监控的文件位置或目录
NotifyFilter	这是 NotifyFilters 枚举值的组合，NotifyFilters 枚举值规定在被监控的文件内要监控哪些内容。这些表示要监控的文件或文件夹的属性。如果规定的属性发生了变化，就引发事件。可能的枚举值是 Attributes、CreationTime、DirectoryName、FileName、LastAccess、LastWrite、Security 和 Size。注意，可以通过二元 OR 运算符来合并这些枚举值
Filter	指定要监控哪些文件的过滤器，例如，*.txt

设置之后，就必须为 4 个事件 Changed、Created、Deleted 和 Renamed 编写事件处理程序。如第 13 章所述，这需要创建自己的方法，并将方法赋给对象的事件。将自己的事件处理程序赋给这些方法，就可以在引发事件时调用方法。当修改与 Path、NotifyFilter 和 Filter 属性匹配的文件或目录

时，就引发每个事件。  
在设置了属性和事件之后，将 `EnableRaisingEvents` 属性设置为 `true`，就可以开始监控工作。下面的示例将在一个简单的客户应用程序中使用 `FileSystemWatcher`，来监控所选目录。

试一试：监控文件系统

- 这是一个较复杂的示例，使用了本章介绍的许多内容。
- (1) 在 `C:\BegVCSharp\Chapter21` 目录中创建一个新的 Windows 应用程序 `FileWatch`。
  - (2) 使用表 21-11 中所示的属性设置各个窗体属性。

表 21-11

属 性	设 置
<code>FormBorderStyle</code>	<code>FixedDialog</code>
<code>MaximizeBox</code>	<code>False</code>
<code>MinimizeBox</code>	<code>False</code>
<code>Size</code>	<code>302, 160</code>
<code>StartPosition</code>	<code>CenterScreen</code>
<code>Text</code>	<code>File Monitor</code>

- (3) 使用表 21-12 中所示的属性，在窗体中添加所需的控件，并设置正确的属性。

表 21-12

控 件	名 称	位 置	大 小	文 本
<code>TextBox</code>	<code>txtLocation</code>	<code>8, 26</code>	<code>184, 20</code>	
<code>Button</code>	<code>cmdBrowse</code>	<code>208, 24</code>	<code>64, 24</code>	<code>Browse...</code>
<code>Button</code>	<code>cmdWatch</code>	<code>88, 56</code>	<code>80, 32</code>	<code>Watch!</code>
<code>Label</code>	<code>lblWatch</code>	<code>8, 104</code>	<code>0, 13</code>	

把 `cmdWatch` 按钮的 `Enabled` 属性设置为 `false`，因为不能在指定文件前就监控它，`lblWatch` 的 `AutoSize` 属性设置为 `true`，以查看其内容。再在窗体上添加一个 `OpenFileDialog` 控件，将其 `Name` 设置为 `FileDialog`，`Filter` 设置为 `All Files|*.*`。结束时，窗体应如图 21-12 所示。

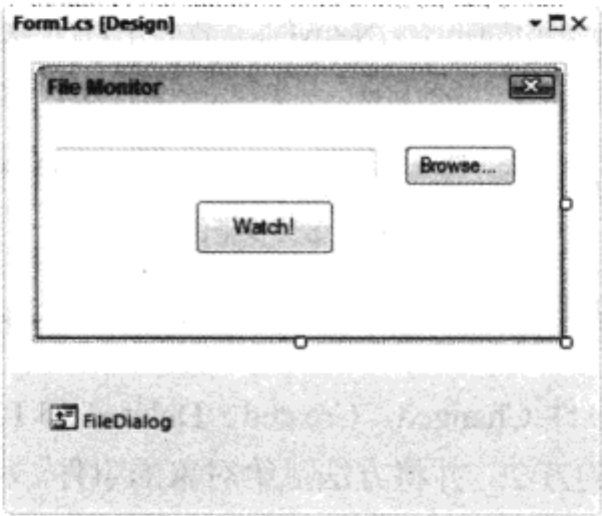


图 21-12

(4) 下面添加一些代码，使窗体可以正常使用。首先需要使用 `using` 指令将 `System.IO` 名称空间添加到已有的 `using` 指令列表中：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
```

代码段 FileWatch\Form1.cs

(5) 在 `Form1` 类中添加 `FileSystemWatcher` 类和一个委托，以便在不同的线程中修改 `lblWatch` 的文本。为此，在 `Form1.cs` 中添加下面的代码：

```
namespace FileWatch
{
    partial class Form1 : Form
    {
        //File System Watcher object.
        private FileSystemWatcher watcher;
        private delegate void UpdateWatchTextDelegate(string newText);
```

(6) 在窗体构造函数中调用 `InitializeComponent()` 方法之后要添加如下代码。这段代码用于初始化 `FileSystemWatcher` 对象，把事件关联到后面要创建的方法上：

```
public Form1()
{
    InitializeComponent();

    this.watcher = new FileSystemWatcher();
    this.watcher.Deleted +=
        new FileSystemEventHandler(this.OnDelete);
    this.watcher.Renamed +=
        new RenamedEventHandler(this.OnRenamed);
    this.watcher.Changed +=
        new FileSystemEventHandler(this.OnChanged);
    this.watcher.Created +=
        new FileSystemEventHandler(this.OnCreate);
}
```

(7) 在 `Form1` 类中添加下面 5 个方法。第一个方法用于在运行 `FileSystemWatcher` 事件处理程序的线程中异步更新 `lblWatch` 中的文本，其他方法是事件处理程序。

```
// Utility method to update watch text.
public void UpdateWatchText(string newText)
{
    lblWatch.Text = newText;
}
```



```
// Define the event handlers.
public void OnChanged(object source, FileSystemEventArgs e)
{
    try
    {
        StreamWriter sw =
            new StreamWriter("C:\\FileLogs\\Log.txt", true);
        sw.WriteLine("File: {0} {1}", e.FullPath,
            e.ChangeType.ToString());
        sw.Close();
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Wrote change event to log");
    }
    catch (IOException)
    {
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Error Writing to log");
    }
}

public void OnRenamed(object source, RenamedEventArgs e)
{
    try
    {
        StreamWriter sw =
            new StreamWriter("C:\\FileLogs\\Log.txt", true);
        sw.WriteLine("File renamed from {0} to {1}", e.OldName,
            e.FullPath);
        sw.Close();
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Wrote renamed event to log");
    }
    catch (IOException)
    {
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Error Writing to log");
    }
}

public void OnDelete(object source, FileSystemEventArgs e)
{
    try
    {
        StreamWriter sw =
            new StreamWriter("C:\\FileLogs\\Log.txt", true);
        sw.WriteLine("File: {0} Deleted", e.FullPath);
        sw.Close();
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Wrote delete event to log");
    }
    catch (IOException)
    {
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Error Writing to log");
    }
}
```

```

public void OnCreate(object source, FileSystemEventArgs e)
{
    try
    {
        StreamWriter sw =
            new StreamWriter("C:\\FileLogs\\Log.txt", true);
        sw.WriteLine("File: {0} Created", e.FullPath);
        sw.Close();
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Wrote create event to log");
    }
    catch (IOException)
    {
        this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
            "Error Writing to log");
    }
}

```

(8) 现在为 Browse 按钮添加 Click 事件处理程序。这个事件处理程序中的代码会打开 Open File 对话框，允许用户选择要监控的文件。双击 Browse 按钮，输入如下代码：

```

private void cmdBrowse_Click(object sender, EventArgs e)
{
    if (FileDialog.ShowDialog() != DialogResult.Cancel)
    {
        txtLocation.Text = FileDialog.FileName;
        cmdWatch.Enabled = true;
    }
}

```

ShowDialog()方法返回表示用户如何退出 File Open 对话框的 DialogResult 枚举值(用户可以单击 OK 按钮或 Cancel 按钮)。我们需要检查是否用户未单击 Cancel 按钮，因此在将用户选择的文件保存到 TextBox 之前，将方法调用的结果与 DialogResult.Cancel 枚举值作比较。最后，将 Watch 按钮的 Enabled 属性设置为 true，以便监控文件。

(9) 双击 Watch 按钮。给 Click 事件处理程序添加下面的代码，以便启动 FileSystemWatcher：

```

private void cmdWatch_Click(object sender, EventArgs e)
{
    watcher.Path = Path.GetDirectoryName(txtLocation.Text);
    watcher.Filter = Path.GetFileName(txtLocation.Text);
    watcher.NotifyFilter = NotifyFilters.LastWrite |
        NotifyFilters.FileName | NotifyFilters.Size;
    lblWatch.Text = "Watching " + txtLocation.Text;
    // Begin watching.
    watcher.EnableRaisingEvents = true;
}

```

(10) 还需要确保 FileLogs 目录是存在的，以便在其中写入数据。在 Form1 构造函数中添加下面的代码，检查该目录是否存在；如果不存在，则创建它。

```

public Form1()
{
    ...
}

```

```
DirectoryInfo aDir = new DirectoryInfo(@"C:\\FileLogs");  
if (!aDir.Exists)  
    aDir.Create();  
}
```

(11) 创建目录 C:\TempWatch，在该目录中创建文件 temp.txt。

(12) 运行应用程序。如果成功地构建了所有内容，则单击 Browse 按钮，并选择 C:\TempWatch\temp.txt。

(13) 单击 Watch 按钮，开始监控文件。在应用程序中可以见到的唯一变化是，标签控件显示正在监控文件。

(14) 使用 Windows Explorer 导航到 C:\TempWatch。在记事本中打开 temp.txt，并在文件中添加一些文本。保存此文件。

(15) 重命名该文件。

(16) 现在可以检查日志文件，以查看其变化。导航到 C:\FileLogs\Log.txt 文件，在记事本中打开它。可以看到对所监控文件的变动说明，如图 21-13 所示。

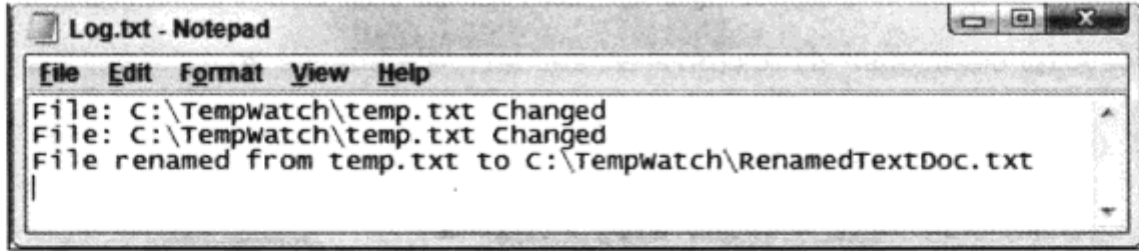


图 21-13

### 示例的说明

此应用程序非常简单，但它演示了 FileSystemWatcher 的工作原理。尝试在监控文本框中输入不同的字符串。如果在目录中指定\*.\*，程序就会监控目录中的所有变化。

应用程序中的大多数代码都用于建立 FileSystemWatcher 对象，以监控正确的位置：

```
watcher.Path = Path.GetDirectoryName(txtLocation.Text);  
watcher.Filter = Path.GetFileName(txtLocation.Text);  
watcher.NotifyFilter = NotifyFilters.LastWrite |  
    NotifyFilters.FileName | NotifyFilters.Size;  
lblWatch.Text = "Watching " + txtLocation.Text;  
// Begin watching.  
watcher.EnableRaisingEvents = true;
```

代码首先设置要监控的目录的路径。这使用了尚未介绍的一个新对象 System.IO.Path。这是一个静态类，非常类似于静态 File 对象。它给出了许多静态方法，以处理和提取文件位置字符串中的信息。这里首先使用它通过 GetDirectoryName()方法提取用户在文本框中键入的目录名称。

下一行代码设置对象的过滤器，过滤器可以是一个实际文件，表示仅监控该文件。过滤器也可以是\*.txt，表示要监控指定目录中的所有.txt 文件。我们也可以使用 Path 静态对象从所提供的文件位置中提取信息。

NotifyFilter 是 NotifyFilters 枚举值的组合，规定组成变化的内容。在此，如果最后写入的时间信息、文件名称或文件大小发生了变化，它就将此变化通知应用程序。在更新了 UI 之后，将

`EnableRaisingEvents` 属性设置为 `true`，开始监控。

但在此之前，还要创建对象，设置事件处理程序。

```
this.watcher = new FileSystemWatcher();
this.watcher.Deleted +=
    new FileSystemEventHandler(this.OnDelete);
this.watcher.Renamed +=
    new RenamedEventHandler(this.OnRenamed);
this.watcher.Changed +=
    new FileSystemEventHandler(this.OnChanged);
this.watcher.Created +=
    new FileSystemEventHandler(this.OnCreate);
```

这就是使用前面创建的私有方法为监控器对象挂接事件处理程序的方式。当删除、重命名、修改或创建文件时，监控器对象就触发事件，调用事件处理程序。用户在自己的方法中可以决定如何处理实际发生的事件。注意，在事件发生之后我们才会得到通知。

在实际的事件处理方法中，只是将事件写入日志文件。显然，根据应用程序的不同，还可以有更复杂的响应。在目录中添加文件时，可以将其移动到别处，或读取其内容，引发新的进程。其可能的用法是无穷无尽的！

## 21.5 小结

本章学习了流和在 .NET Framework 中使用流访问文件和其他序列化设备的原因。我们介绍了 `System.IO` 名称空间中的基类，包括：

- `File`
- `FileInfo`
- `FileStream`

`File` 类提供了许多静态方法，用于移动、复制和删除文件，`FileInfo` 表示磁盘上的一个物理文件，其方法可以处理该文件。`FileStream` 对象表示只读、只写或读写的文件。我们还介绍了 `StreamReader` 和 `StreamWriter` 类，以及它们在写入流时的作用。学习了使用 `FileStream` 类读写随机文件的方法。在此基础上，使用 `System.IO.Compression` 名称空间中的类在把流写入磁盘时压缩流，以及把对象序列化到文件中。最后构建了一个完整的应用程序，使用 `FileSystemWatcher` 类监控文件和目录。

概括地讲，本章学习了：

- 打开文件，读取文件，写入文件
- `StreamWriter` 和 `StreamReader` 类与 `FileStream` 类的区别
- 使用带分隔符的文件填充数据结构
- 压缩和解压缩流
- 序列化和反序列化对象
- 使用 `FileSystemWatcher` 类监控文件系统

21.6 练习

- (1) 只有导入哪个名称空间才允许应用程序使用文件？
- (2) 何时使用 FileStream 对象，而不是使用 StreamWriter 对象写入文件？
- (3) StreamReader 类的哪些方法允许从文件中读取数据，每个方法的具体作用是什么？
- (4) 哪个类可使用 Deflate 算法压缩流？
- (5) 如何阻止所创建的类型序列化？
- (6) FileSystemWatcher 类提供了哪些事件，其作用是什么？
- (7) 修改本章构建的 FileWatch 应用程序。无需退出应用程序就可以打开和关闭文件系统监控功能。

附录 A 给出了练习答案。

21.7 本章要点

主 题	重 要 概 念
流	流是序列化设备的一种抽象表示，可以一次从序列化设备中读取或写入一个字节。文件就是这种设备的一个例子。流有两种类型：输入和输出，分别用于读取和写入设备
文件访问类	.NET Framework 中有大量抽象了文件系统访问的类，包括通过静态方法处理文件和目录的 File 和 Directory，可实例化为表示特定文件和目录的 FileInfo 和 DirectoryInfo。后两个类在对文件和目录执行多个操作时使用，因为这两个类不要求为每个方法调用指定路径。可以在文件和目录上执行的典型操作包括查看和修改属性，创建、删除和复制操作
文件路径	文件和目录路径可以是绝对的或相对的。绝对路径给出了某位置的完整描述，从包含它的驱动器的根目录开始。所有的父目录都与子目录用反斜杠隔开。相对路径与之类似，但从文件系统的指定点开始，例如执行应用程序的目录(工作目录)。浏览文件系统时，常常使用父目录的别名..
FileStream 对象	FileStream 对象允许访问文件的内容，以进行读写。它以字节为单位访问文件数据，所以并不总是访问文件数据的最佳选项。FileStream 实例维护着文件内部的一个位置字节索引，这样就可以浏览文件的内容了。以这种方式访问文件的任意位置称为随机访问
读写流	一种读写文件数据的更简单方法是使用 StreamReader 和 StreamWriter 类，以及 FileStream。它们允许读写字符和字符串数据，而不是处理字节。这些类型提供了我们熟悉的处理字符串的方法，包括 ReadLine()和 WriteLine()。因为它们处理的是字符串数据，所以使用这些类，可以更加方便地处理逗号分隔的文件(这是表示结构化数据的常见方式)
压缩文件	可以使用 DeflateStream 和 GZipStream 压缩流类读写文件中的压缩数据。这些类与 FileStream 一样，也处理字节数据，但可以通过 StreamReader 和 StreamWriter 类访问数据，以简化代码
对象序列化	我们常常需要存储和检索表示对象状态的数据。可以使用序列化技术来自动保存和加载对象状态，而无需编写代码来保存和加载属性值。为此，必须使用 Serializable 特性把对象类型标记为可序列化。还可以使用其他属性控制如何序列化成员，例如 NonSerialized 特性禁止序列化给定的成员
监控文件系统	可以使用 FileSystemWatcher 类监控文件系统数据的变化。可以监控文件和目录，如果需要，还可以提供一个过滤器，根据需要仅修改有特定扩展名的文件。FileSystemWatcher 实例通过触发事件，来通知我们发生了变化，这些变化可以在代码中处理

# 第 22 章

## XML

### 本章内容:

---

- 如何读写 XML
- 应用于格式良好的 XML 的规则
- 如何根据两种模式 XSD 和 XDR 验证 XML 文档
- 如何在应用程序中使用 XML
- 如何在程序中通过 .NET 使用 XML
- 如何使用 XPath 查询搜索 XML 文档

Extensible Markup Language(可扩展标记语言, XML)近几年得到了广泛的关注。XML 并不是新技术,肯定不是 Microsoft 为了在 .NET 环境中使用而开发的,但 Microsoft 在 XML 的开发初期就认识到了它的潜力,所以,XML 在 .NET 中执行大量的任务,包括描述应用程序的配置、在 Web 服务之间传输信息等。

XML 是一种以简单文本格式存储数据的方式,这意味着它可以被任何计算机读取。在前面章节介绍 Web 编程技术时可以看到,XML 是在 Internet 上传输数据的绝佳格式。甚至对于人们来说,它也很容易理解!

从 VS 的第一版开始,Microsoft 就在开发使用 XML 的解决方案上投入了大量的精力。目前 .NET 中的大多数应用程序都以某种形式使用 XML,例如存储配置细节的 .config 文件, WPF 中使用的 XAML 文件等。甚至 Office 2007 引入的新文档格式都基于 XML,虽然 Office 应用程序本身并不是 .NET 应用程序。

XML 的细节非常复杂,因此在此不介绍其所有细节。但其基本格式非常简单,大多数情况下,不需要了解 XML 的详细知识,因为 VS 通常会处理其中大多数工作——我们基本上不必手动编写 XML 文档。前面已经说过,XML 在 .NET 领域非常重要,因为它是传输数据的默认格式,所以理解其基本知识至关重要。



## 22.1 XML 文档

XML 中的完整数据集就是 XML 文档。XML 文档可以是计算机上的物理文件，或是内存中的字符串。但是其本身必须是完整的，必须遵循某些规则(稍后介绍)。XML 文档由许多不同的部分组成。其中最重要的部分是 XML 元素，它包含文档的实际数据。

### 22.1.1 XML 元素

XML 元素包含一个开始标记(放在尖括号中的元素名称，如<myElement>)、元素中的数据和结束标记(与开始标记相同，但是在左括号后有一个斜线：</myElement>)。

例如，定义一个存储书名的元素：

```
<book>Tristram Shandy</book>
```

如果您了解一些 HTML 知识，就可能认为这非常类似——非常正确！事实上，HTML 和 XML 有许多相同的语法。最大的不同是，XML 没有任何预定义元素——我们选择自定义元素的名称，因此元素数量不受限制。最重要的是 XML——不管它使用什么样的名称——实际上不是语言，而是定义语言的标准(称为 XML 应用)。每种语言都有自己独特的词汇库——一组可以用于文档的特定元素和允许采用这些元素的结构。稍后会介绍，我们可以显式地限制 XML 文档中允许使用的元素。另外，也可以使用任何元素，允许使用文档的程序给出其结构。

元素名称区分大小写，因此<book>和<BOOK>是不同的元素。这表示，如果试图通过使用大小写不同的结束标记(例如</BOOK>)关闭<book>元素，XML 文档就是非法的。XML 分析程序(parser)读取 XML 文档，并分析其中各个元素，它们会拒绝任何包含非法 XML 的文档。

元素也可以包含其他元素，因此可以修改此<book>元素，添加两个子元素，使之包括作者和标题：

```
<book>
  <title>Tristram Shandy</title>
  <author>Lawrence Sterne</author>
</book>
```

但是元素不允许重叠，因此在父元素的结束标记之前必须关闭所有子元素。例如，不能编写如下代码：

```
<book>
  <title>Tristram Shandy
  <author>Lawrence Sterne
  </title></author>
</book>
```

这是非法的，因为在<title>元素内打开了<author>元素，但是结束标记</title>位于结束标记</author>之前。

所有元素必须有结束元素的规定有一个例外。可以有“空”元素，其中没有内嵌的数据或文本。在此，可以直接在开始元素之后添加结束标记，如下所示：

```
<book></book>
```



或者使用简短语法，在开始元素末尾添加结束元素的斜线：

```
<book />
```

### 22.1.2 特性

除了在元素体内存储数据之外，也可以在特性内存储数据，将特性添加到元素的开始标记内。特性的形式为：

```
name="value"
```

其中特性值必须包含在单引号或双引号内。例如：

```
<book title="Tristram Shandy"></book>
```

或者：

```
<book title='Tristram Shandy'></book>
```

这都是合法的，但是下面的语句不合法：

```
<book title=Tristram Shandy></book>
```

为什么在 XML 中需要两种方式来存储数据？下面二者的区别是什么：

```
<book>
  <title>Tristram Shandy</title>
</book>
```

和

```
<book title="Tristram Shandy"></book>
```

实际上，二者并没有太大的区别。使用其中任何一个都没有什么优势可言。如果以后需要为数据添加更多信息，最好选择使用元素——总是可以给元素添加子元素或特性，但是对特性就不能进行这样的操作。有争议的是，元素是否更易于读取，更简洁(这只能根据个人不同的爱好来决定)。另一方面，如果未经压缩就在网络上传输文档，则特性会占用更少的带宽(即使压缩了，区别也不大)，更便于保存对文档的每一位用户而言无关紧要的信息。也许最好的选择是同时使用二者，可以根据自己的爱好选择使用某种方式来存储特定的数据项。但是确实没有硬性规定。

### 22.1.3 XML 声明

除了元素和特性之外，XML 文档还可以包含许多组成部分。XML 文档的各个组成部分称为节点——因此元素、元素内的文本和特性都是 XML 文档的节点。如果需要深入研究 XML，这些都是非常重要的。但是只有一种类型的节点存在于几乎所有的 XML 文档中。这就是 XML 声明，如果包括了它，它就必须是文档的第一个节点。

XML 声明的格式类似于元素，但是在尖括号内有问号。它的名称始终都是 xml，并总是有 version 特性；当前，它只有两个值：1.0(第一版)和 1.1(第二版)，但 VS 不支持第二版。第二版并没有给 Windows 平台上使用的 XML 做什么改进，而且 World Wide Web Consortium ([www.w3c.org](http://www.w3c.org))建议尽可能使用第一版。因此最简单的 XML 声明形式为：

```
<?xml version="1.0"?>
```

另外，它还可以包含特性 `encoding`(其值表示用于读取文档的字符集，比如 UTF-16 表示文档使用 16 位 Unicode 字符集)和 `standalone`(其值是 `yes` 或 `no`，表示 XML 文档是否依赖于其他文件)。但是这些特性并不是必需的，可以仅在 XML 文件中包括 `version` 特性。

#### 22.1.4 XML 文档的结构

XML 最重要的一点就是它提供了一种组织数据的结构化方式，它非常不同于关系数据库。最新型的数据库系统在表中存储数据，表通过各列中的值而相互关联。每个表在行和列中存储数据——每一行代表一个记录，每一列代表该记录的特定数据项。相反，XML 数据是分层组织的，有点类似于 Windows Explorer 中的文件夹和文件。每个文档必须有一个根元素，其中包含所有元素和文本数据。如果在文档的顶级中有多个元素，该文档就是不合法的 XML 文档。但是可以在顶级包括其他 XML 节点——通常是 XML 声明。所以下面的 XML 文档是合法的：

```
<?xml version="1.0"?>
<books>
  <book>Tristram Shandy</book>
  <book>Moby Dick</book>
  <book>Ulysses</book>
</books>
```

但是下面的文档就不合法：

```
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
```

在此根元素之下，组织数据时具有极大的灵活性。在关系数据中，每一行都有相同的列数，而元素所包含的子元素数量不受限制。尽管 XML 文档经常以与关系数据类似的方式组织，每一个记录具有一个元素，但 XML 文档不需要任何预定义的结构。这是传统关系数据库和 XML 的主要区别之一。关系数据库总是在添加数据之前定义信息的结构，而信息存储在 XML 中时不需要这个初始设置，所以 XML 是存储小块数据的非常便利的方式。稍后就会看到，可以为 XML 提供结构，但与关系数据库不同，这个结构不是强制的，除非明确要求有这个结构。

#### 22.1.5 XML 名称空间

如第 9 章所述，每个人都可以定义自己的 C# 类，也可以定义自己的 XML 元素；这就导致了一个问题——如何知道哪个元素属于哪个词汇表？注意本节的标题，可以按类似的方式回答这个问题。如同定义名称空间以组织 C# 类型一样，可以使用 XML 名称空间定义 XML 词汇表。这就可以将不同词汇表中的元素包含到一个 XML 文档中，而没有因为(例如)两个不同的词汇表定义了一个 `<customer>` 元素而误解元素的风险。

XML 名称空间非常复杂，因此在此不详细介绍它，但其基本语法非常简单。使用前缀，后跟冒号，就可以将具体的元素或特性关联到特定的名称空间。例如，`<wrox:book>` 表示 `wrox` 名称空间中的 `<book>` 元素。但是如何知道名称空间 `wrox` 所表示的内容呢？为了使此方法有效，必须保证每个名称空间都是唯一的。最简单的办法是将前缀映射到某些独特事物上。如下所示：在 XML 文档

的某个位置,需要将名称空间前缀关联到 Uniform Resource Identifier(唯一资源标识符,URI)上。URI 包含几种类型,最常见的类型是 Web 地址,如 `www.wrox.com`。

为了用具体的名称空间标识前缀,可以在元素内使用 `xmlns:prefix` 特性,将其值设置为标识名称空间的唯一 URI。然后前缀就可以用于该元素的任何位置,包括任何内嵌的子元素。例如:

```
<?xml version="1.0"?>
<books>
  <book xmlns:wrox="http://www.wrox.com">
    <wrox:title>Beginning C#</wrox:title>
    <wrox:author>Karli Watson</wrox:author>
  </book>
</books>
```

在此, `<title>` 和 `<author>` 元素使用 `wrox:` 前缀,因为它们位于 `<book>` 元素内,其中定义了该前缀。但是如果试图将此前缀添加到 `<books>` 元素上,XML 就是非法的,因为并没有为此元素定义前缀。

也可以使用 `xmlns` 特性为元素定义默认的名称空间:

```
<?xml version="1.0"?>
<books>
  <book xmlns="http://www.wrox.com">
    <title>Beginning Visual C#</title>
    <author>Karli Watson</author>
    <html:img alt="Cover Image" src="begcsharp.gif"
      xmlns:html="http://www.w3.org/1999/xhtml" />
  </book>
</books>
```

这里将 `<book>` 元素的默认名称空间定义为 `"http://www.wrox.com"`。因此此元素内的所有内容都属于此名称空间,除非添加不同的名称空间前缀显式指定其他内容,比如对 `<img>` 元素所做的处理(将它设置到与 XML 兼容的 HTML 文档所使用的名称空间中)。

### 22.1.6 格式良好并有效的 XML

前面介绍的都是“合法”的 XML。事实上,有两种合法形式的 XML: 格式良好和有效。遵循 XML 标准的所有规则的文档就是“格式良好(well-formed)”的 XML。如果 XML 文档的格式有误,分析程序就不能正确地解释它,并拒绝该文档。为了有一个格式良好的 XML,对文档的要求如下:

- 有且只有一个根元素
- 每一个元素都有结束标记(上面提到的简短语法除外)
- 没有重叠元素——所有子元素必须完全嵌套在父元素内
- 所有特性必须放在引号内

这些还不够全面,但强调了 XML 新手容易犯的一些错误。完全遵循这些规则的 XML 文档仍然可能是无效的。前面说过,XML 本身不是语言,而是定义 XML 应用程序的标准。格式良好的 XML 文档仅仅符合 XML 标准;为了使之有效,它们还必须符合对 XML 应用程序所规定的规则。并不是所有的分析程序都检查文档是否有效;进行这种检查的分析程序就是验证分析程序。但是为了检查一个文档是否符合应用程序的规则,首先需要一种指定规则的方式。

## 22.1.7 验证 XML 文档

XML 支持通过两种方法，来定义在文档中可以放置那些元素和特性，及其放置顺序——文档类型定义(Document Type Definitions, DTD)和模式。

### 1. DTD

DTD 使用从 XML 的父文档继承的非 XML 语法，并逐渐被模式(schema)所代替。DTD 不允许规定元素和属性的数据类型，因此不太灵活，在 .NET Framework 的环境中用得不多。另一方面，模式使用得非常多——它们允许规定数据类型，是用 XML 兼容的语法编写的。但是，模式非常复杂，有不同的格式定义它们——即使在 .NET 中也是如此！

### 2. 模式

.NET 支持的模式具有两种不同的格式——XML Schema Definition 语言(XSD)和 XML-Data Reduced 模式(XDR)。

#### XDR 模式

XDR 模式定义是一个旧标准，专用于 Microsoft，除了 Microsoft 分析器外，其他分析器通常不能识别它，也很少使用它。XSD 是一个开放标准，W3C 推荐这个模式，所以这里介绍其定义。模式可以包括在 XML 文档内，也可以放在单独文件中。在编写 XML 文档之前需要非常熟悉 XML。但是，在模式中识别主要元素时它非常有用，因此我们要解释一下基本原则。为此，介绍这个简单 XML 文档的示例 XSD 模式，它包含了两本 Wrox 的 C# 图书的详情。这个 XML 在本书的下载代码中，其文件名是 book.xml:



```
<?xml version="1.0"?>
<books>
  <book>
    <title>Beginning Visual C# 2010</title>
    <author>Karli Watson</author>
    <code>7582</code>
  </book>
  <book>
    <title>Professional C# 2010</title>
    <author>Simon Robinson</author>
    <code>7043</code>
  </book>
</books>
```

Code snippet book.xml

#### XSD 模式

XSD 模式中的元素必须属于名称空间 <http://www.w3.org/2001/XMLSchema>。如果未包括此名称空间，就不能识别这些模式元素。

为了将 XML 文档与另一文件中的 XSD 模式关联，需要在根元素中添加 `schemalocation` 元素：

```
<?xml version="1.0"?>
```

```
<books schemalocation="file:///C:/Beginning Visual C#/Chapter 22/books.xsd">
.
</books>
```

下面分析一个 XSD 模式示例:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="books">
    <complexType>
      <choice maxOccurs="unbounded">
        <element name="book">
          <complexType>
            <sequence>
              <element name="title" />
              <element name="author" />
              <element name="code" />
            </sequence>
          </complexType>
        </element>
      </choice>
      <attribute name="schemalocation" />
    </complexType>
  </element>
</schema>
```

在此首先要注意,默认名称空间设置为 XSD 名称空间。这就告诉分析器,文档中的所有元素属于模式。如果不指定此名称空间,分析器就认为元素仅是普通的 XML 元素,认识不到需要使用它们进行验证。

完整的模式包含在<schema>元素内(使用小写 s——大小写非常重要!)。文档中的每个元素都必须由<element>元素来表示。此元素具有指示元素名称的 name 特性。如果元素将要包含嵌套的子元素,就必须在<complexType>元素内为这些子元素包含<element>标记。可以在其中指定子元素的操作方式。

例如,使用<choice>元素指定对子元素进行选择操作,或使用<sequence>规定子元素必须以它们在模式中列举的顺序出现。如果一个元素可能多次出现(如<book>元素),就需要在其父元素内包括 maxOccurs 特性。将其设置为 unbounded,表示元素可以出现无限次。最后,特性必须由<attribute>元素表示,包括 schemalocation 特性,它告诉分析器模式所在的位置。它放在子元素列表的结尾处。

前面介绍了 XML 的基本理论,接下来的示例创建 XML 文档。VS 为我们完成了大部分繁重工作,甚至基于 XML 文档创建了 XSD 模式,我们无需编写任何代码!

#### 试一试: 在 VS 中创建 XML 文档

按照下面的步骤,创建 XML 文档。

(1) 打开 VS,从菜单中选择 File | New | File。如果没有看到这个选项,请创建一个新项目。在 Solution Explorer 中右击项目,选择添加一个新项。然后从对话框中选择 XML File。

(2) 在 New File 对话框中,选择 XML File,单击 Add。VS 会自动创建一个新的 XML 文档。如图 22-1 所示,VS 添加了 XML 声明,以 encoding 特性结束(其特性和元素也是彩色的,但是在黑白纸张中其效果不太好)。

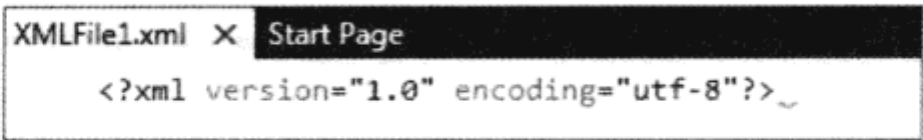


图 22-1

(3) 按下 Ctrl+S 组合键，或者在菜单中选择 File | Save XMLFile1.xml，保存文件。VS 会询问文件的保存位置，以及文件的名称。将其保存在 BegVCSharp\Chapter22 文件夹中，取名为 GhostStories.xml。

(4) 将光标移动到 XML 声明下面的代码行，键入文本<stories>。注意，输入大于号关闭开始标记时，VS 会自动置入结束标记。

(5) 输入下面的 XML 文件，单击 Save:



```
<stories>
  <story>
    <title>A House in Aungier Street</title>
    <author>
      <name>Sheridan Le Fanu</name>
      <nationality>Irish</nationality>
    </author>
    <rating>eerie</rating>
  </story>
  <story>
    <title>The Signalman</title>
    <author>
      <name>Charles Dickens</name>
      <nationality>English</nationality>
    </author>
    <rating>atmospheric</rating>
  </story>
  <story>
    <title>The Turn of the Screw</title>
    <author>
      <name>Henry James</name>
      <nationality>American</nationality>
    </author>
    <rating>a bit dull</rating>
  </story>
</stories>
```

Code snippet Chapter22\GhostStories.xml

(6) 现在可以让 Visual Studio 为刚才编写的 XML 文件创建相应的模式。为此，从 XML 菜单中选择 Create Schema 菜单项，单击 Save as GhostStories.xsd，保存得到的 XSD 文件。

(7) 返回到 XML 文件，在结束标记</stories>之前键入如下 XML:

```
<story>
  <title>Number 13</title>
  <author>
    <name>M.R. James</name>
    <nationality>English</nationality>
```



```
</author>
<rating>mysterious</rating>
</story>
```

注意, 开始键入开始标记时, 会显示 IntelliSense 提示。这是因为 Visual Studio 知道把新建的 XSD 模式连接到正在键入的 XML 文件上。

(8) 可以在 Visual Studio 中创建 XML 和一个或多个模式之间的链接。选择 XML | Schemas, 会打开如图 22-2 所示的对话框。在 Visual Studio 可识别的长长模式列表顶部, 会看到 GhostStories.xsd。在它的左边是一个绿色的复选标记, 表示这个模式用于当前的 XML 文档。

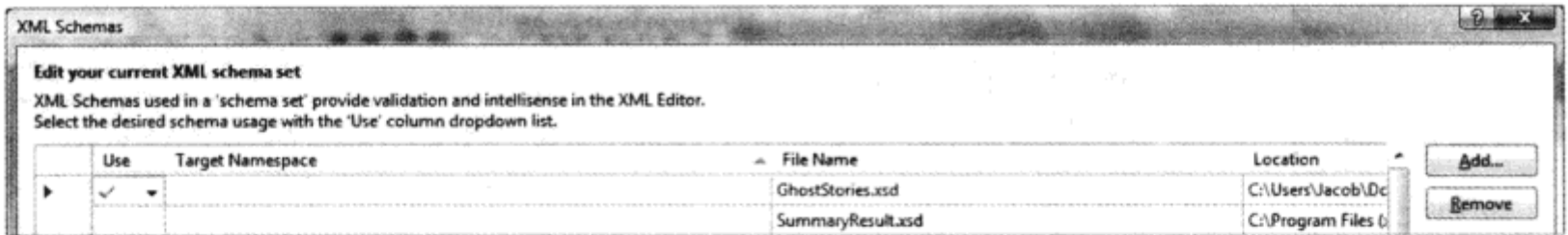


图 22-2



图 22-2 显示的 XSD 对话框包含 VS 可识别的一个很长的模式列表, 但它不会自动存储以前使用的模式。如果重复使用一个模式, 不希望每次在需要它时都指定它, 就可以把它复制到如下位置 C:\Program Files\Microsoft Visual Studio 10.0\Xml\Schemas。复制到这个位置的所有模式都会显示在 Schemas 对话框中。

## 22.2 在应用程序中使用 XML

了解如何创建 XML 文档后, 就应把这些知识用于实践。 .NET Framework 提供了许多名称空间和类, 使 XML 的读取、处理和写入非常简单。下面几节将介绍这些类, 说明如何使用它们以编程方式创建和处理 XML。

### 22.2.1 XML 文档对象模型

XML 文档对象模型(Document Object Model, DOM)是一组以非常直观的方式访问和处理 XML 的类。DOM 不是读取 XML 数据的最快捷的方式, 但只要理解了类和 XML 文档中元素之间的关系, DOM 就很容易使用。

构成 DOM 的类在名称空间 System.Xml 中。在这个名称空间中有几个类和子名称空间。但本章只介绍几个易于操作 XML 的类。要学习和使用的类如表 22-1 所示。

表 22-1

类 名	说 明
XmlNode	这个类表示文档树中的一个节点, 它是本章许多类的基类。如果这个节点表示 XML 文档的根, 就可以从它导航到文档的任意位置



(续表)

类 名	说 明
XmlDocument	扩展了 XmlNode 类，但常常是使用 XML 的第一个对象，因为这个类用于加载磁盘或其他地方的数据并在这些位置保存数据
XmlElement	表示 XML 文档中的一个元素。XmlElement 派生于 XmlLinkedNode，XmlLinkedNode 派生于 XmlNode
XmlAttribute	表示一个特性，与 XmlDocument 类一样，它也派生于 XmlNode 类
XmlText	表示开始标记和结束标记之间的文本
XmlComment	表示一种特殊类型的节点，这种节点不是文档的一部分，但为阅读器提供文档各部分的信息
XmlNodeList	表示一个节点集合

1. XmlDocument 类

通常，要处理 XML 的应用程序，首先应从磁盘中读取它。如表 22-1 中所示，这是 XmlDocument 类的工作。可以将 XmlDocument 看作磁盘上文件的内存表示。使用 XmlDocument 类把文件加载到内存中后，就可以从中获得文档的根节点，开始读取和处理 XML 了：

```
using System.Xml;
.
.
.

XmlDocument document = new XmlDocument();
document.Load(@"C:\Beginning Visual C#\Chapter 22\books.xml");
```

这两行代码创建了 XmlDocument 类的一个新实例，并在其中加载 books.xml 文件。Xml Document 类位于 System.Xml 名称空间中，所以应在代码开头的 using 部分插入 using System.Xml; 语句。

除了加载和保存 XML 之外，XmlDocument 类还负责维护 XML 结构。所以，这个类有许多方法可以用于创建、修改和删除树中的节点。稍后将介绍其中一些方法，但为了正确理解这些方法，还需要了解另一个类：XmlElement。

2. XmlElement 类

文档加载到内存中后，就要对它执行一些操作。上面代码创建的 XmlDocument 实例的 DocumentElement 属性会返回一个 XmlElement 实例(表示 XmlDocument 的根节点)。这个元素非常重要，因为有了它，就可以访问文档中的所有信息。

```
XmlDocument document = new XmlDocument();
document.Load(@"C:\Beginning Visual C#\Chapter 22\books.xml");
XmlElement element = document.DocumentElement;
```

获得文档的根节点后，就可以使用信息了。XmlElement 类包含的方法和属性可以处理树的节点和特性。下面首先看看用于导航 XML 元素的属性，如表 22-2 所示。

表 22-2

属 性	说 明
FirstChild	<p>这个属性返回当前节点之后的第一个子节点。在本章前面的 books.xml 文件中，文档的根节点是 books，根节点之后的节点是 book，在该文档中，根节点 books 的第一个子节点是 book。</p> <pre>&lt;books&gt;    @@1a Root node   &lt;book&gt;    @@1a Firstchild</pre> <p>nodeFirstChild 返回一个 XmlNode 对象，应测试返回节点的类型，因为它不总是一个 XmlElement 实例。在 books 示例中，Title 元素的子元素是表示文本 Beginning Visual C# 的 XmlText 节点</p>
LastChild	<p>这个属性的操作与 FirstChild 属性十分类似，但返回当前节点的最后一个子节点。在 books 示例中，books 节点的最后一个子节点仍是 book，但它表示 Professional C# 2010。</p> <pre>&lt;books&gt; @@1a Root node   &lt;book&gt; @@1a FirstChild     &lt;title&gt;Beginning Visual C# 2010&lt;/title&gt;     &lt;author&gt;Karli Watson&lt;/author&gt;     &lt;code&gt;7582&lt;/code&gt;   &lt;/book&gt;   &lt;book&gt; @@1a LastChild     &lt;title&gt;Professional C# 2010&lt;/title&gt;     &lt;author&gt;Simon Robinson&lt;/author&gt;     &lt;code&gt;7043&lt;/code&gt;   &lt;/book&gt; &lt;/books&gt;</pre>
ParentNode	<p>这个属性返回当前节点的父节点。在 books 示例中，books 节点是 book 节点的父节点</p>
NextSibling	<p>FirstChild 和 LastChild 属性返回当前节点的叶子节点，而 NextSibling 节点返回有相同父节点的下一个节点。在 books 示例中，title 元素的 NextSibling 属性返回 author 元素，在 author 元素上调用 NextSibling，会返回 code 元素</p>
HasChildNodes	<p>检查当前元素是否有子元素，而无需获取 FirstChild 的值并检查它是否为 null</p>

使用表 22-2 中的 4 个属性，可以遍历整个 XmlDocument，如下面的示例所示。

试一试：迭代 XML 文档中的所有节点

在这个示例中，要创建一个小型 Windows Forms 应用程序，迭代 XML 文档中的所有节点，打印出元素的名称，如果是 XmlText 元素，就打印出包含在元素中的文本。这段代码使用了 book.xml，如前面的“模式”一节所述。如果没有创建这个文件，可在本书的下载代码中找到它。

(1) 首先创建一个新的 Windows 窗体项目。选择 File | New | Project 菜单项，在打开的对话框中，选择 Windows | Windows Forms Application。将项目命名为 LoopThroughXmlDocument，按下回车键。

(2) 将一个 TextBox 和一个 Button 控件拖放到窗体上，按照图 22-3 所示设计窗体。

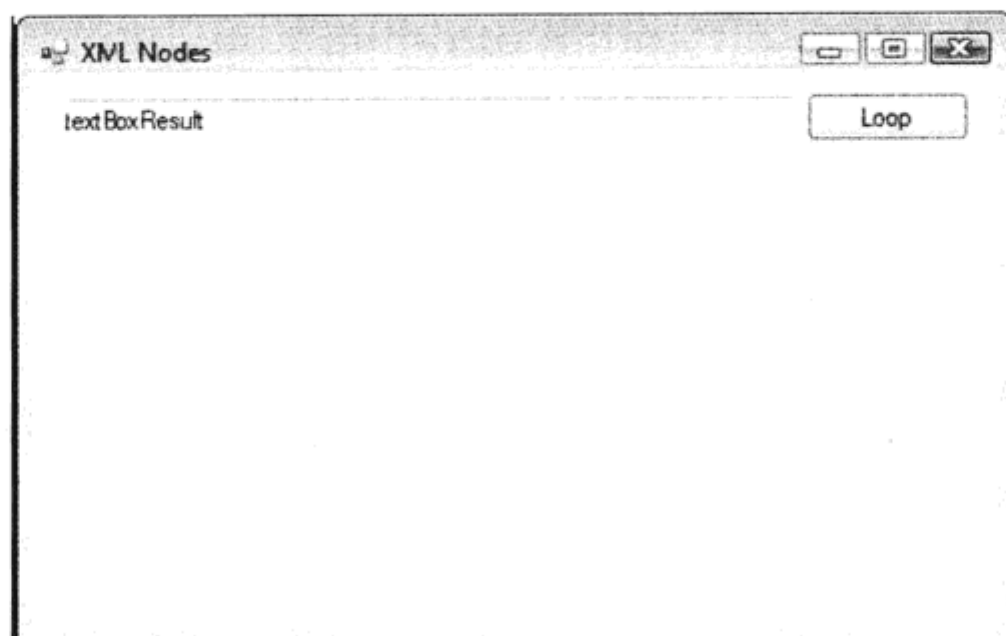


图 22-3

(3) 将文本框命名为 `textBoxResult`，按钮命名为 `buttonLoopThroughDocument`。把文本框的 `Multiline` 属性设置为 `true`，`Scrollbars` 属性设置为 `Vertical`。

(4) 双击按钮，输入下面的代码。注意，要在文件顶部的 `using` 部分添加 `using System.Xml;`

```
private void buttonLoopThroughDocument_Click(object sender, EventArgs e)
{
    XmlDocument document = new XmlDocument();
    document.Load(@"C:\Beginning Visual C# 2010\Chapter 22\Books.xml");
    textBoxResult.Text = FormatText(document.DocumentElement as XmlNode, "", "");
}

private string FormatText(XmlNode node, string text, string indent)
{
    if (node is XmlText)
    {
        text += node.Value;
        return text;
    }

    if (string.IsNullOrEmpty(indent))
        indent = "";
    else
    {
        text += "\r\n" + indent;
    }

    if (node is XmlComment)
    {
        text += node.OuterXml;
        return text;
    }

    text += "<" + node.Name;
    if (node.Attributes.Count > 0)
    {
        AddAttributes(node, ref text);
    }
}
```

```

if (node.HasChildNodes)
{
    text += ">";
    foreach (XmlNode child in node.ChildNodes)
    {
        text = FormatText(child, text, indent + " ");
    }
    if (node.ChildNodes.Count == 1 &&
        (node.FirstChild is XmlText || node.FirstChild is XmlComment))
        text += "</" + node.Name + ">";
    else
        text += "\r\n" + indent + "</" + node.Name + ">";
}
else
    text += " />";
return text;
}

private void AddAttributes(XmlNode node, ref string text)
{
    foreach (XmlAttribute xa in node.Attributes)
    {
        text += " " + xa.Name + "=" + xa.Value + "'";
    }
}
}

```

---

Code snippet Chapter22\LoopThroughXmlDocument\Form1.cs

---

(5) 运行该应用程序，单击 Loop 按钮。结果如图 22-4 所示。

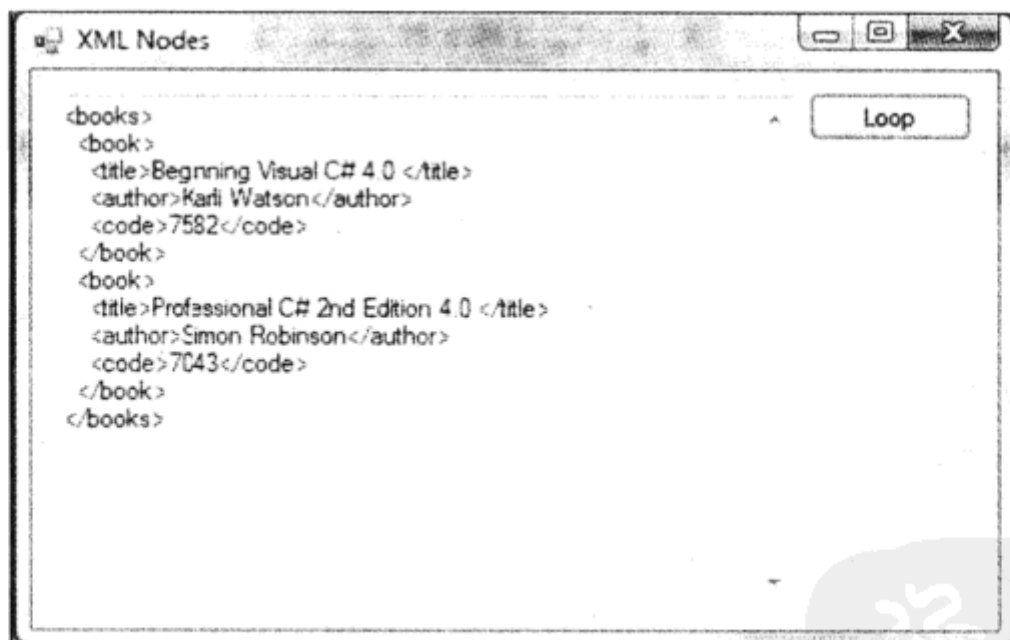


图 22-4

#### 示例的说明

单击按钮时，首先会调用 XmlDocument 方法 Load。这个方法把文件中的 XML 加载到 XmlDocument 实例中，XmlDocument 实例用于访问 XML 的元素。接着调用一个方法来递归迭代 XML，并把 XML 文档的根节点传送给方法。根元素是使用 XmlDocument 类的属性 DocumentElement 获得的。除了在传送给 RecurseXmlDocument 方法的根参数上检查 null 之外，还要注意 if 语句：

```
if (node is XmlText)
{
    .
}
}
```

is 运算符可以检查对象的类型，如果实例是指定的类型，就返回 true。即使根节点声明为 XmlNode，这也只是要操作的对象的基本类型。使用 is 运算符除了可以测试对象的类型之外，还可以在运行期间确定对象类型，并根据该类型选择要执行的操作。

在 FormatText 方法中给文本框生成文本。注意必须知道根节点的当前实例的类型，因为要显示的信息的获取方式对于不同的元素来说是不同的。我们要显示 XmlElements 的名称和 XmlText 元素的值。

3. 修改节点的值

在了解如何改变节点值之前，先要明白，节点值一般比较复杂。实际上，即使派生于 XmlNode 的所有类都包含 Value 属性，它也很少返回有用的信息。初看起来它可能令人失望，但实际上是十分合理的。看看前面的 books 示例：

```
<books>
  <book>
    <title>Beginning Visual C# 4.0</title>
    <author>Karli Watson</author>
    <code>7582</code>
  </book>
  <book>
</books>
```

文档中的每对标记都解析为 DOM 中的一个节点。在迭代文档中的所有节点时，会遇到许多 XmlElement 节点和三个 XmlText 节点。上述 XML 中的 XmlElement 节点是<books>、<book>、<title>、<author>和<code>。XmlText 节点是 title、author 和 code 开始标记和结束标记之间的文本。title、author 和 code 的值是标记之间的文本，但文本本身就是一个节点，节点包含了值。其他标记都没有相关的值。

在上述 FormatText 方法的代码靠近顶部的位置，if 块中的下述代码在当前节点是 XmlText 时执行：

```
text += node.Value;
```

XmlText 节点实例的 Value 属性用于获取节点的值。

如果使用 XmlElement 类型的节点的 Value 属性，就返回 null，但如果使用另外两个方法 InnerText 和 InnerXml 中的一个，就可以获取 XmlElement 开标记和闭标记之间的信息。也就是说，可以使用两个方法和一个属性来操作节点的值，如表 22-3 所示。

表 22-3

属 性	说 明
InnerText	这个属性获取当前节点中所有子节点的文本，把它作为一个串联字符串返回。也就是说，在上面的 XML 中，如果获取 book 节点的 InnerText 值，就返回字符串 Beginning Visual C 2010#Karli Watson7582。如果获取 title 节点的 InnerText，就只返回 Beginnning Visual C# 2010。 可以使用这个方法设置文本，但要小心，因为如果设置了错误节点的文本，就很可能改写不想改变的信息

(续表)

属 性	说 明
InnerXml	InnerXml 属性返回类似于 InnerText 的文本，也返回所有的标记。如果获取 book 节点上的 InnerXml 值，结果是如下字符串：  <title>Beginning Visual C# 2010</title><author>Karli Watson</author><code>7582</code>  可以看出，如果字符串包含要直接插入 XML 文档的内容，这是很有用的。但是要对该字符串负全责，如果插入格式错误的 XML，应用程序就会生成异常
Value	Value 属性是操作文档中信息的最精练方式，但如前所述，在获取值时，只有几个类会返回有用的信息。返回所需文本的类如下所示：  XmlText  XmlComment  XmlAttribute

插入新节点

了解了如何遍历 XML 文档，如何获取元素的值后，下面学习如何给前面使用的 books 文档添加节点，改变文档的结构。

要在列表中插入新元素，需要使用 XmlDocument 和 XmlNode 类中的新方法，如表 22-4 所示。XmlDocument 类的方法可以创建新的 XmlNode 和 XmlElement 实例，这非常不错，因为这两个类都只有一个受保护的构造函数，不能直接使用 new 创建它们的实例。

表 22-4

方 法	说 明
CreateNode	创建任意类型的节点。该方法有三个重载版本，其中两个允许创建 XmlNodeType 枚举中所列出的类型的节点，另一个允许把要使用的节点类型指定为字符串。除非对指定的不是枚举中的节点类型有完全的把握，否则强烈推荐使用枚举的两个重载版本。该方法返回的一个 XmlNode 实例，该实例可以显式转换为合适的类型
CreateElement	这只是 CreateNode 的一个版本，只能创建 XmlElement 类型的节点
CreateAttribute	这也只是 CreateNode 的一个版本，只能创建 XmlAttribute 类型的节点
CreateTextNode	创建 XmlTextNode 类型的节点
CreateComment	在这个列表中包含这个方法，是为了说明可以创建的节点类型的多样性。这个方法并不创建由 XML 文档表示的数据节点，而是创建注释，以便人们读取数据。在应用程序中读取文档时，就可以读取注释

表 22-4 中的方法都用于创建节点，在调用其中一个方法后，就必须执行一些操作。在创建节点后，节点并没有包含其他信息，节点也没有插入到文档中。为此，应使用派生于 XmlNode 的类(包括 XmlDocument 和 XmlElement)中的方法。表 22-5 描述了这些方法。

表 22-5

方 法	说 明
AppendChild	把一个子节点追加到 XmlNode 类型或其派生类型的节点上。在调用该方法之后，追加的节点显示在相应节点的子节点列表的最后。如果不关心子节点的顺序，这就不重要，但如果子节点的顺序很重要，就应以正确的顺序追加节点
InsertAfter	使用 InsertAfter 方法，可以控制插入新节点的位置。该方法带有两个参数，第一个是新节点，第二个是在其后插入新节点的节点
InsertBefore	这个方法与 InsertAfter 类似，但新节点插到参考节点之前

下面的示例以前面的示例为基础，在 books.xml 文档中插入一个 book 节点。该示例中没有清理文档的代码，所以如果该示例运行几次，文档中就会包含许多相同的节点。

试一试：创建节点

按照下面的步骤给 books.xml 文档添加一个节点。

- (1) 在窗体的已有按钮下面添加一个按钮，命名为 buttonCreateNode。将其Text 属性改为 Create Node。
- (2) 双击新按钮，输入下面的代码：



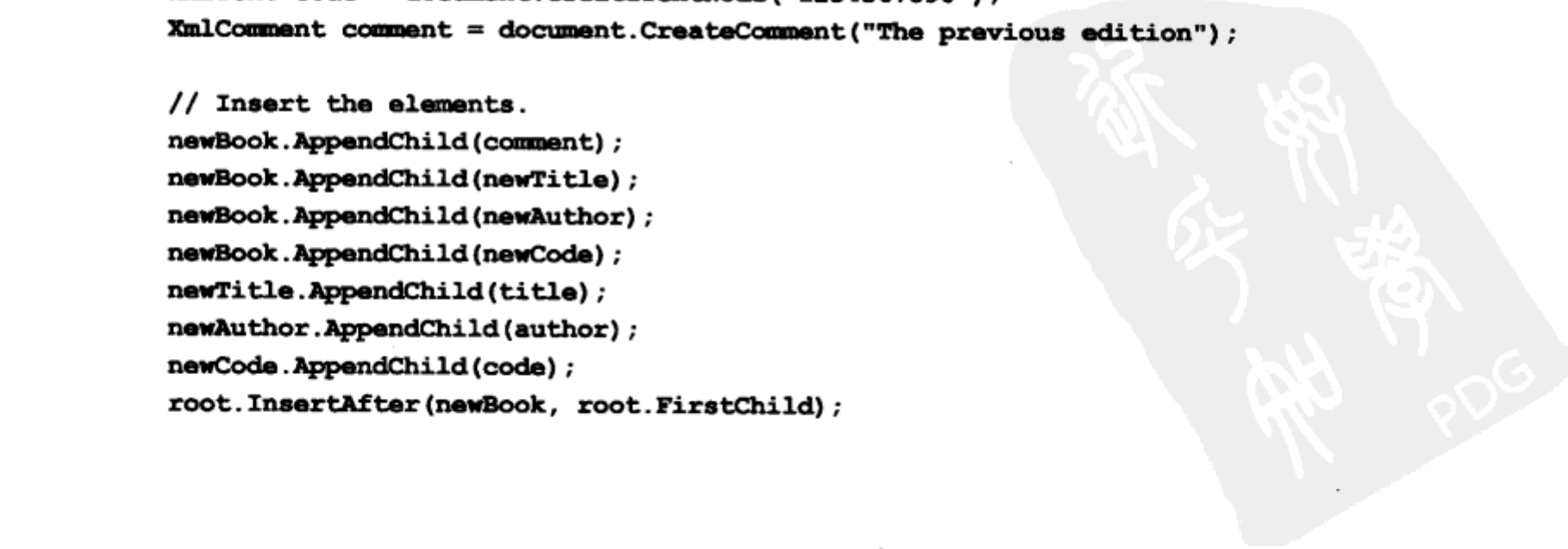
可从  
wrox.com  
下载源代码

```
private void buttonCreateNode_Click(object sender, EventArgs e)
{
    // Load the XML document.
    XmlDocument document = new XmlDocument();
    document.Load(@"C:\Beginning Visual C#\Chapter 22\Books.xml");

    // Get the root element.
    XmlElement root = document.DocumentElement;

    // Create the new nodes.
    XmlElement newBook = document.CreateElement("book");
    XmlElement newTitle = document.CreateElement("title");
    XmlElement newAuthor = document.CreateElement("author");
    XmlElement newCode = document.CreateElement("code");
    XmlText title = document.CreateTextNode("Beginning Visual C# 2008");
    XmlText author = document.CreateTextNode("Karli Watson et al");
    XmlText code = document.CreateTextNode("1234567890");
    XmlComment comment = document.CreateComment("The previous edition");

    // Insert the elements.
    newBook.AppendChild(comment);
    newBook.AppendChild(newTitle);
    newBook.AppendChild(newAuthor);
    newBook.AppendChild(newCode);
    newTitle.AppendChild(title);
    newAuthor.AppendChild(author);
    newCode.AppendChild(code);
    root.InsertAfter(newBook, root.FirstChild);
}
```





```
document.Save(@"C:\Beginning Visual C#\Chapter 22\Books.xml");
}
```

Code snippet Chapter22\InsertingNodes\Form1.cs

(3) 运行应用程序，单击 Create Node 按钮。再单击 Loop 按钮，对话框如图 22-5 所示。

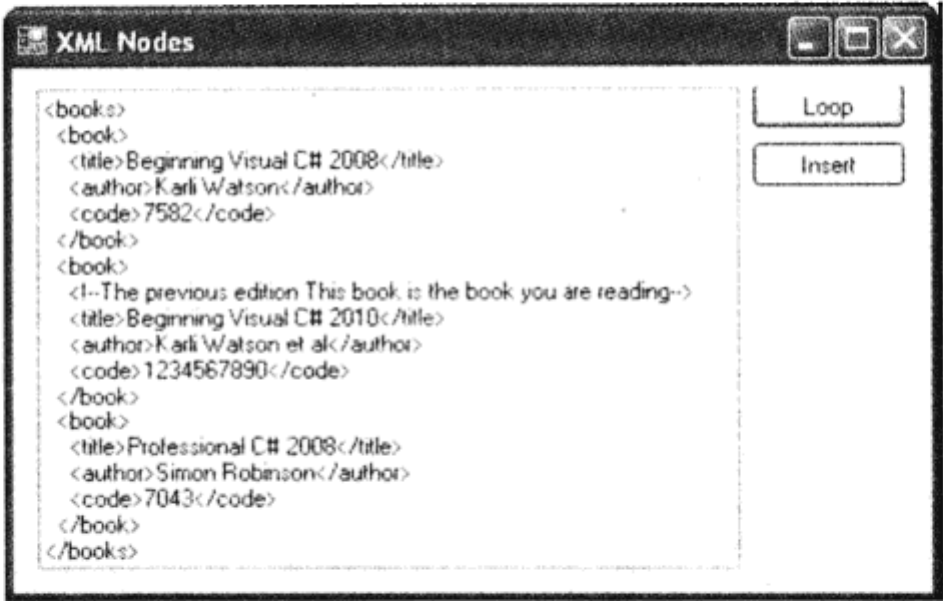


图 22-5

在上面的示例中没有创建一个重要类型的节点：XmlAttribute，这里把它留作本章的练习。

示例的说明

buttonCreateNode\_Click 方法中的代码创建了所有节点。它创建了 8 个新节点，其中 4 个节点的类型是 XmlElement，3 个节点的类型是 XmlText，最后一个节点的类型是 XmlComment。

所有节点都是使用封装 XmlDocument 实例的方法创建的。XmlElement 节点是用 CreateElement 方法创建的，XmlText 节点是用 CreateTextNode 方法创建的，XmlComment 节点是用 CreateComment 方法创建的。

创建完节点后，还需要把它们插入 XML 树。这是使用元素上的 AppendChild 方法实现的，新节点将成为该元素的一个子节点。唯一的例外是book 节点，它是所有新节点的根节点。这个节点使用根对象的 InsertAfter 方法插入到树中。使用 AppendChild 方法插入的所有节点总是成为子节点列表的最后一项，而 InsertAfter 允许指定节点的位置。

删除节点

学习了如何创建新节点，剩下的就是如何删除节点了。派生于XmlNode 的所有类都包含允许从文档中删除节点的两个方法，如表 22-6 所示。

表 22-6

方 法 名	说 明
RemoveAll	这个方法删除节点上的所有子节点。不太明显的是，它还会删除节点上的所有特性，因为也将它们看作子节点
RemoveChild	这个方法删除节点上的一个子节点，返回从文档中删除的节点。如果改变主意，还可以把它重新插回到文档

下面的示例将扩展前面两个示例创建的 Windows 窗体应用程序，使之包含删除节点的功能。只需找出 book 节点的最后一个实例，并删除它。

### 试一试：删除节点

下面的步骤将找出 book 节点的最后一个实例，并删除它。

(1) 在前面两个已有的按钮下面添加一个新按钮，命名为buttonDeleteNode，将其 Text 属性设置为 Delete Node。

(2) 双击新按钮，输入下面的代码：



可从  
wrox.com  
下载源代码

```
private void buttonDeleteNode_Click(object sender, EventArgs e)
{
    // Load the XML document.
    XmlDocument document = new XmlDocument();
    document.Load(@"C:\Beginning Visual C# 2010\Chapter 22\Books.xml");

    // Get the root element.
    XmlElement root = document.DocumentElement;

    // Find the node. root is the <books> tag, so its last child
    // which will be the last <book> node.
    if (root.HasChildNodes)
    {
        XmlNode book = root.LastChild;

        // Delete the child.
        root.RemoveChild(book);

        // Save the document back to disk.
        document.Save(@"C:\Beginning Visual C# 2010\Chapter 22\Books.xml");
    }
}
```

代码段 Chapter22\RemovingNodes\Form1.cs

(3) 运行应用程序。单击 Delete Node 按钮，再单击 Loop 按钮，树中的最后一个节点就会消失。

#### 示例的说明

把 XML 加载到 XmlDocument 对象上后，就检查根元素，确定在加载的 XML 中是否有子元素。如果有，就使用 XmlElement 类的 LastChild 属性获取最后一个子元素。之后，调用 RemoveChild，给它传送要删除的元素实例(在本例中是根元素的最后一个子元素)，就删除了该元素。

### 22.2.2 选择节点

前面介绍了如何浏览 XML 文档、如何操作文档的值、如何创建新节点以及如何删除它们。剩下的就是在不遍历整个树的情况下选择节点。


XmlNode 类包含的两个方法常用于从文档中选择节点，且不遍历其中的每个节点，如表 22-7 所示。这两个方法是 SelectSingleNode 和 SelectNodes，它们都使用一种特殊的查询语言 XPath 来选择节点。稍后介绍该语言。

表 22-7

方 法	说 明
SelectSingleNode	选择一个节点。如果创建一个查找多个节点的查询，就只返回第一个节点
SelectNodes	以 XmlNodeList 类的形式返回一个节点集合

22.2.3 XPath

XPath 是 XML 文档的查询语言，就像 SQL 是关系数据库的查询语言一样。它由表 22-7 中的两个方法使用，以免遍历 XML 文档的整个树。但是需要花一定的时间才能熟悉它，因为其语法与 SQL 或 C#完全不同。



XPath 相当复杂，这里只介绍其中的一小部分，就足以选择节点了。如果想了解 XPath 的更多内容，可参阅 [www.w3.org/TR/xpath](http://www.w3.org/TR/xpath) 和 Visual Studio 帮助页面。

为了正确使用 XPath，下面要使用 XML 文件 Elements.xml。该文档包含元素周期表中的部分化学元素。这个 XML 文件的部分内容列在本章后面“选择节点”示例中，其完整内容可以在本书网站的本章下载代码 Elements.xml 中找到。

表 22-8 列出了 XPath 执行的最常见操作。如果未特别说明，XPath 查询示例就根据它操作的节点来选择。必须有一个节点名称，可以假定当前节点是上述 XML 文档中的<book>节点。

表 22-8

目 的	XPath 查询示例
选择当前节点	.
选择当前节点的父节点	..
选择当前节点的所有子节点	*
选择带有特定名称的所有子节点，这里是 title	title
选择当前节点的一个特性	@Type
选择当前节点的所有特性	@*
按照索引选择一个子节点，这里是第二个元素节点	element[2]
选择当前节点的所有文本节点	text()
选择当前节点的一个或多个孙子节点	element/text()
在文档中选择带有特定名称的所有节点，在这里是所有的 mass 节点	//mass
在文档中选择带有特定名称和特定父节点名称的所有节点，在这里父节点名称是 element，节点名称是 name	//element/name
选择值满足条件的节点，在这里，选择元素名为 Hydrogen 的元素	//element[name='Hydrogen']
选择特性值满足条件的节点，在此，Type 特性的值是 Noble Gas	//element[@Type='Noble Gas']

下面的“试一试”示例要创建一个小型应用程序，以执行许多预定义的查询，查看结果，并可以输入自己的查询。

## 试一试：选择节点

如前所示，这个示例使用新的 XML 文件 Element.xml。可以从本书的网站上下载这个文件，或者键入下面的代码：



可从  
Wrox.com  
下载源代码

```
<?xml version="1.0"?>
<elements>
  <!--First Non-Metal-->
  <element Type="Non-Metal">
    <name>Hydrogen</name>
    <symbol>H</symbol>
    <number>1</number>
    <specification>
      <mass>1.007825</mass>
      <density>0.0899 g/cm3</density>
    </specification>
  </element>
  <!--First Noble Gas-->
  <element Type="Noble Gas">
    <name>Helium</name>
    <symbol>He</symbol>
    <number>2</number>
    <specification>
      <mass>4.002602</mass>
      <density>0.1785 g/cm3</density>
    </specification>
  </element>
  <!--First Halogen-->
  <element Type="Halogen">
    <name>Fluorine</name>
    <symbol>F</symbol>
    <number>9</number>
    <specification>
      <mass>18.998404</mass>
      <density>1.696 g/cm3</density>
    </specification>
  </element>
  <element Type="Noble Gas">
    <name>Neon</name>
    <symbol>Ne</symbol>
    <number>10</number>
    <specification>
      <mass>20.1797</mass>
      <density>0.901 g/cm3</density>
    </specification>
  </element>
</elements>
```

代码段 Chapter22\XpathQuery\Elements.xml

把这个 XML 文件保存为 Elements.xml。一定要修改下述代码中文件的路径。这个示例是一个小型查询工具，可用于测试通过代码提供的 XML 上的不同查询。

按照下面的步骤创建一个带有查询功能的 Windows 窗体应用程序。

- (1) 创建一个新的 Windows 窗体应用程序，命名为 XPathQuery。
- (2) 创建如图 22-6 所示的对话框。按照图中所示给控件命名，但按钮除外，它应命名为 `buttonExecute`，把 `textBoxResult` 控件的 `Scrollbars` 属性设置为 `Vertical`。

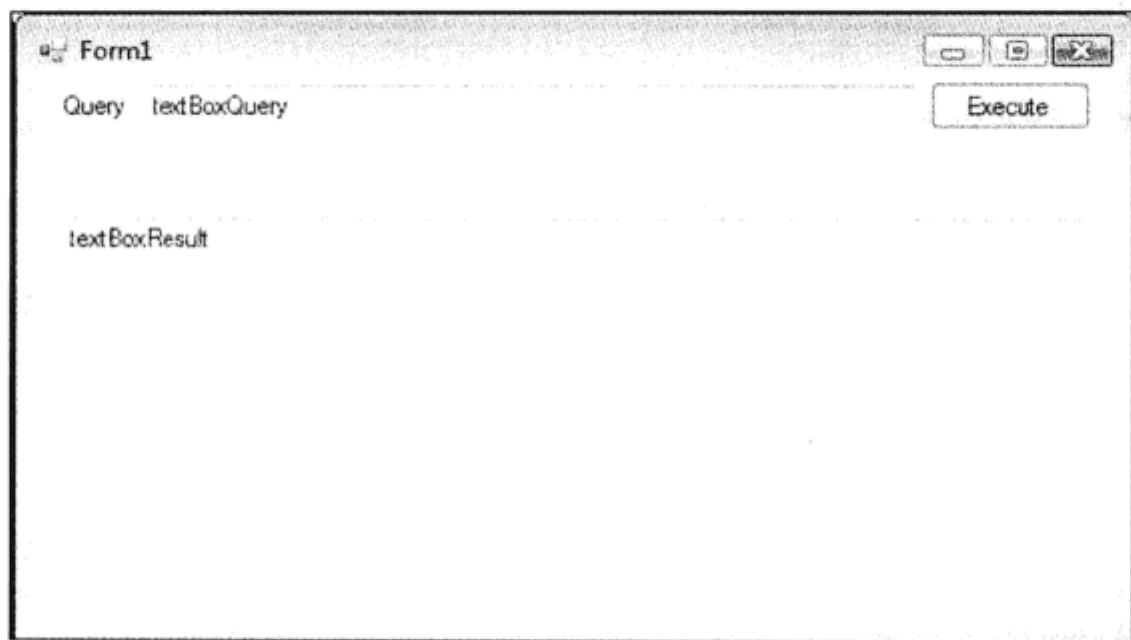


图 22-6

- (3) 右击窗体，选择 `View Code`。首先包含 `using` 指令：

```
using System.Xml;
```

- (4) 接着，添加一个私有字段，来保存文档，在构造函数中初始化它：

```
private XmlDocument document;

public Form1()
{
    InitializeComponent();

    document = new XmlDocument();
    document.Load(@"C:\Beginning Visual C#\Chapter 22\Elements Subset.xml");
}
```

- (5) 在此需要使用几个帮助方法，以便在文本框 `textBoxResult` 中显示查询结果：

```
private void Update(XmlNodeList nodes)
{
    if (nodes == null || nodes.Count == 0)
    {
        textBoxResult.Text = "The query yielded no results";
        return;
    }
    string text = "";
    foreach (XmlNode node in nodes)
    {
        text = FormatText(node, text, "") + "\r\n";
    }
    textBoxResult.Text = text;
}
```

(6) 更新构造函数，以便在应用程序启动时显示 XML 文件的全部内容：

```
public Form1()
{
    InitializeComponent();

    document = new XmlDocument();
    document.Load(@"C:\Beginning Visual C#\Chapter 22\Elements Subset.xml");

    Update(document.DocumentElement.SelectNodes("."));
}
```

(7) 把上一个“试一试”示例中的 FormatText 和 AddAttributes 方法复制粘贴到新项目中。

(8) 最后插入代码，执行用户在文本框中输入的内容：

```
private void buttonExecute_Click(object sender, EventArgs e)
{
    try
    {
        XmlNodeList nodes = document.DocumentElement.SelectNodes(textBoxQuery.Text);
        Update(nodes);
    }
    catch (Exception err)
    {
        textBoxResult.Text = err.Message;
    }
}
```

(9) 运行应用程序，在 textBoxQuery 文本框中输入下面的查询，以选择一个元素节点，其中包含文本为 Hydrogen 的节点。

```
element[name=' Hydrogen' ]
```

#### 示例的说明

buttonExecute\_Click 方法是执行查询的方法。因为我们不可能事先知道键入到 textBoxQuery 中的查询会生成一个还是多个节点，所以必须使用 SelectNodes 方法。该方法返回一个 XmlNodeList 对象，但如果所使用的查询是非法的，该方法就抛出一个与 XPath 相关的异常。

Update 方法负责遍历 SelectNodes 选择出来的 XmlNodeList 的内容。它对每个节点调用前面示例中的 FormatText，FormatText 负责递归遍历节点树，创建可以在 textBoxResult 控件中读取的文本。

在本章最后的练习中，读者需要执行许多其他的 XPath 查询。在把它们输入 XPathQuery 应用程序中查看结果时，尝试自己确定查询结果。

## 22.3 小结

本章学习了 Extensible Markup Language (XML)，XML 是一种存储和提取数据的文本格式。我们讨论了需要遵循的规则，以确保 XML 文档是格式良好的。还论述了如何根据 XSD 和 XDR 模式验证它们。

学习了 XML 的基础知识后，探讨了如何使用 C# 和 Visual Studio 在代码中使用 XML，最后阐述了如何使用 XPath 在 XML 中进行查询。

第 23 章将学习如何使用一种非常有趣的查询语言 LINQ。这种语言也可以用于查询 XML。但这超

出了本书的范围。在此之前，请先完成下面的练习。

22.4 练习

- (1) 修改“创建节点”示例中的插入方式，把值为 1000 的特性 Pages 插入 book 节点。
- (2) 确定下述 XPath 查询的结果，再把查询键入“选择节点”示例中的 XPathQuery 应用程序，来验证自己的结果。注意所有的查询都在元素节点 DocumentElement 上执行。

```
//elements
element
element[@Type=' Noble Gas' ]
//mass
//mass/..
element/specification[mass=' 20.1797' ]
element/name[text()=' Neon' ]
```

- (3) 在许多 Windows 系统中，XML 的默认查看器都是 Web 浏览器。如果使用 Internet Explorer，在其中加载 Elements.xml 文件，就会看到美观的 XML 的格式化视图。在浏览器控件(而不是文本框)中显示查询的 XML 的效果为什么不理想？

附录 A 给出了练习答案。

22.5 本章要点

主 题	重 要 概 念
XML 语法	XML 文档从 XML 声明、XML 名称空间、XML 元素和特性中创建。XML 声明定义了 XML 版本，XML 名称空间用于定义词汇表，XML 元素和特性用于定义 XML 文档的内容
格式良好的 XML	格式良好的 XML 是遵循 XML 基本语法规则的 XML。如果文档正好有一个根元素，每个元素都有一个结束标记，没有相互重叠的元素(所有的子元素都必须完全嵌套在父元素中)，所有的特性都放在引号中，该文档就是格式良好的。所有 XML 读取器都可以读取格式良好的 XML，很少允许读取格式有误的 XML。严格地说，如果结束标记的文档是格式有误的，它就不是 XML 文档
有效的 XML	有效的 XML 是格式良好的 XML，且通过检查 XML 是从模式中生成的，就可以验证 XML 的有效性。确保 XML 有效是非常重要的，因为这样可以对 XML 文档的内容做一些假定，如果知道文档中的结构和名称与期望的一致，就可以处理利用这些结构和名称生成的 XML
XML 模式	XML 模式用于定义 XML 文档的结构。需要与第三方交换信息时，模式特别有用。对所交换的数据的模式达成一致后，我们和第三方就可以检查该文档是否有效
程序中的 XML	目前 XML 在.NET 领域中广泛使用，.NET Framework 为创建和处理 XML 提供了一系列类。可以使用 XML 存储应用程序配置，把数据保存到磁盘上，在线上传送信息等
XPath	XPath 是在 XML 文档中查询数据的一种方式之一。要使用 XPath，就必须熟悉 XML 文档的结构，才能从中选择各个元素。尽管 XPath 可以用于任何格式良好的 XML 文档，但创建查询时必须了解文档的结构，这意味着确保文档有效，也就确保了只要文档对于同一个模式而言是有效的，查询就可以用于多个文档





# 第 23 章

## LINQ 简介

### 本章内容:

---

- 编写 LINQ 查询, LINQ 查询语言的组成部分
- 使用 LINQ 方法语法和 LINQ 查询语法
- 对查询结果排序, 包括多级排序
- 使用 LINQ 聚合运算符的方式和场合
- 使用投射(Projection)在查询中创建新对象
- 使用 Distinct()、Any()、All()、First()、FirstOrDefault()、Take()和 Skip()运算符
- 组合查询
- Set 运算符和联合

本章介绍 Language Integrated Query(LINQ), 这是 C# 3.0 语言引入的一个扩展, C#3.0 是 Visual Studio 2010 中支持的 C# 4 语言的上一版。LINQ 可以处理非常大的数据集合, 这一般需要选择集合的一个子集, 来完成执行程序的任务。

过去, 完成这类任务需要编写大量的循环代码, 而额外的处理甚至需要更多的代码, 例如, 排序或组合所找到的对象。而 LINQ 使我们无需编写这些循环代码, 就可以进行过滤和排序操作。这样我们就可以集中关注程序中的重要对象。

除了提供一种简洁的查询语言, 以便您精确指定要搜索的对象之外, LINQ 还提供了许多扩展方法, 更便于排序、组合和计算查询结果的统计数据。

使用 LINQ 可以查询 C#中许多不同的数据源, 包括对象、SQL 数据库、XML 文档、实体数据模型和外部应用程序(如 Amazon Web 服务和公司目录)。本章介绍的 LINQ 语法和方法适用于所有不同的数据源。查询不同数据源的 LINQ 提供程序在下一章介绍。

LINQ非常大, 完整地介绍它的所有特性和方法已超出了本书的讨论范围。但本章将列举 LINQ 用户需要的所有运算符和语句类型的示例, 并酌情给出深入探讨这些主题的资料。

## 23.1 第一个 LINQ 查询

下面先介绍一个示例。这个示例使用 LINQ 创建了一个查询，以便在一个简单的内存对象数组中查找一些数据，并输出到控制台上。

### 试一试：第一个 LINQ 查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-1-FirstLINQquery，然后打开主源文件 Program.cs。

(2) 注意，Visual C# 2010 默认在 Program.cs 中包含 Linq 名称空间：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

(3) 在 Program.cs 的 Main()方法中添加如下代码：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
                      "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

    var queryResults =
        from n in names
        where n.StartsWith("S")
        select n;

    Console.WriteLine("Names beginning with S: ");

    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue: ");
    Console.ReadLine();
}
```

代码段 23-1-FirstLINQquery\Program.cs

(4) 编译并运行程序(按下 F5 键即可开始调试)，列表中的名称以 S 开头，按照它们在数组中的声明顺序排列，如下所示。

```
Names beginning with S:
Smith
Smythe
Small
```

```
Singh
Samba
Program finished, press Enter/Return to continue:
```

按下回车键，结束程序，关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能)，就需要按下回车键两次，这会结束程序的运行。

### 示例的说明

第一步是引用 System.Linq 名称空间，这在创建项目时由 Visual C# 2010 自动完成：

```
using System.Linq;
```

所有的基本底层系统都支持 System.Linq 名称空间中用于 LINQ 的类。如果在 Visual C# 2010 外部创建 C#源文件或编辑以前的 Visual C# 2005 项目，就必须手动添加 using System.Linq 语句。

下一步创建一些数据，在本例中就是声明并初始化 names 数组：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
    "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
```

这些数据很少，很适用于查询结果比较明显的示例。程序的下一部分是真正的 LINQ 查询语句：

```
var queryResults =
    from n in names
    where n.StartsWith("S")
    select n;
```

这是一个看起来比较古怪的语句。它不像是 C#语言，实际上 from..where..select 语法类似于 SQL 数据库查询语言。但这个语句不是 SQL，而是 C#，在 Visual C# 2010 中输入这些代码时，from、where 和 select 会突出显示为关键字，这个古怪的语法对编译器而言是完全正确的。

这个程序中的 LINQ 查询语句使用了 LINQ 声明性查询语法：

```
var queryResults =
    from n in names
    where n.StartsWith("S")
    select n;
```

该语句包括 4 个部分：以 var 开头的结果变量声明，使用查询表达式给该结果变量赋值，查询表达式包含 from 子句、where 子句和 select 子句。下面逐一介绍它们。

#### 23.1.1 用 var 关键字声明结果变量

LINQ 查询首先声明一个变量，以包含查询的结果，这通常是用 var 关键字声明一个变量来完成的：

```
var queryResults =
```

如第 14 章所述，var 是 C#中的一个新关键字，用于声明一般的变量类型，特别适合于包含 LINQ 查询的结果。var 关键字告诉 C#编译器，根据查询推断结果的类型。这样，就不必提前声明从 LINQ 查询返回的对象类型了——编译器会推断出该类型。如果查询返回多个条目，该变量就是查询数据源中的一个对象集合(在技术上它并不是一个集合，只是看起来像是集合而已)。



如果希望了解细节, 查询结果将是实现了 `IEnumerable<>` 接口的类型。 `IEnumerable` 后面的尖括号表示这是一个泛型类型。泛型详见第 12 章。

在上例中, 编译器创建了 `System.Linq.OrderedSequence<string, string>` 的一个实例, 这个特殊的 LINQ 数据类型提供了字符串的有序列表(因为数据源是一个字符串集合)。

另外, `queryResult` 名称是随意指定的, 可以把结果命名为任何名称, 例如, `namesBeginningWithS` 或者在程序中有意义的其他名称。

### 23.1.2 指定数据源: from 子句

LINQ 查询的下一部分是 `from` 子句, 它指定了要查询的数据:

```
from n in names
```

本例中的数据源是前面声明的字符串数组 `names`。变量 `n` 只是数据源中某一元素的代表, 类似于 `foreach` 语句后面的变量名。指定 `from` 子句, 就可以只查找集合的一个子集, 而不用迭代所有的元素。

说到迭代, LINQ 数据源必须是可枚举的——即必须是数组或集合, 以便从中选择一个或多个元素。



从技术上讲, 这表示数据源必须支持 `IEnumerable<>` 接口, 所有的 C# 数组或项目集合都支持这个接口。

数据源不能是单个值或对象, 例如, 单个 `int` 变量。如果只有一项, 就没有必要查询了。

### 23.1.3 指定条件: where 子句

在 LINQ 查询的下一部分, 可以用 `where` 子句指定查询的条件, 如下所示:

```
where n.StartsWith("S")
```

可以在 `where` 子句中指定能应用于数据源中各元素的任意布尔(`true` 或 `false`)表达式。实际上, `where` 子句是可选的, 甚至可以忽略, 但在大多数情况下, 都要指定 `where` 条件, 把结果限制为我们需要的数据。`where` 子句称为 LINQ 中的限制运算符, 因为它限制了查询的结果。

这个示例指定 `name` 字符串以字母 `S` 开头, 还可以给字符串指定其他条件, 例如, 长度超过 10(`where n.Length > 10`)或者包含 `Q`(`where n.Contains("Q")`)。

### 23.1.4 指定元素: select 子句

最后, `select` 子句指定结果集中包含哪些元素。`select` 子句如下所示:

```
select n;
```

`select` 子句是必须的, 因为必须指定结果集中有哪些元素。这个结果集并不是很有趣, 因为在结

果集的每个元素中都只有一项 `name`。如果结果集中有比较复杂的对象，使用 `select` 子句的有效性就比较明显，不过我们还是先完成这个示例。

### 23.1.5 完成：使用 `foreach` 循环

现在输出查询的结果。与把数组用作数据源一样，像这样的 LINQ 查询结果是可以枚举的，即可以用 `foreach` 语句迭代结果：

```
Console.WriteLine("Names beginning with S:");

foreach (var item in queryResults) {
    Console.WriteLine(item);
}
```

在本例中，匹配了 4 个名称：`Singh`、`Small`、`Smythe` 和 `Samba`，所以它们会显示在 `foreach` 循环中。

### 23.1.6 延迟执行的查询

`foreach` 循环实际上并不是 LINQ 的一部分，它只是迭代结果。虽然 `foreach` 结构并不是 LINQ 的一部分，但它是实际执行 LINQ 查询的代码。查询结果变量仅保存了执行查询的一个计划，在访问查询结果之前，并没有提取 LINQ 数据，这称为查询的延迟执行或迟缓执行。生成结果序列(即列表)的查询都要延迟执行。

现在回过头来看看代码。由于输出了结果，所以程序结束：

```
Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
```

这些代码仅确保在按下一个键(甚至可以按下 `F5` 键，而不是 `Ctrl+F5` 组合键)之前，控制台程序的结果始终显示在屏幕上。在大多数其他 LINQ 示例中也使用这种结构。

## 23.2 使用 LINQ 方法语法

用 LINQ 完成同一任务有多种方式，但常常需要通过编程来实现。如前所述，前面的示例是用 LINQ 查询语法编写的，下一个示例是用 LINQ 的方法语法(也称为显式语法，但这里使用“方法语法”这个术语)编写的相同程序。

### 23.2.1 LINQ 扩展方法

LINQ 实现为一系列扩展方法，用于集合、数组、查询结果和其他实现了 `IEnumerable` 接口的对象。在 Visual Studio IntelliSense 特性中可以看到这些方法。例如，在 Visual C# 2010 中打开 `FirstLINQQuery` 程序中的 `Program.cs` 文件，在 `name` 数组的下面键入对该数组的一个新引用：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
    "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

names.
```

键入 `names` 后面的句点后, 就会看到 Visual Studio IntelliSense 列出的可用于 `names` 的方法。

`Where<>` 方法和大多数其他方法都是扩展方法(在 `Where<>` 方法的右边显示了一个文档说明, 它以 `extension` 开头)。因为如果在顶部注释掉了指令 `using System.Linq`, `Where<>`、`Union<>`、`Take<>` 和大多数其他方法就会从列表中消失。上一个示例使用的 `from..where..select` 查询表达式由 C# 编译器转换为这些方法的一系列调用。使用 LINQ 方法语法时, 就直接调用这些方法。

### 23.2.2 查询语法和方法语法

查询语法是在 LINQ 中编写查询的首选方式, 因为它一般更容易理解, 最常见的查询使用它们也更简单。但是, 一定要基本了解方法语法, 因为一些 LINQ 功能不能通过查询语法来使用, 或者使用方法语法比较简单。



Visual C# 2010 联机帮助建议尽量使用查询语法, 仅在需要时使用方法语法。

本章主要使用查询语法, 但会指出需要方法语法的场合, 并说明如何使用方法语法来解决问题。

大多数使用方法语法的 LINQ 方法都要求传送一个方法或函数, 来计算查询表达式。方法/函数参数以委托的形式传送, 它一般引用一个匿名方法。

LINQ 很容易完成这个传送任务。使用 Lambda 表达式就可以创建方法/函数。参见第 14 章。下一个示例将更清楚地说明这一点。

#### 试一试: 使用 LINQ 方法语法

按照下面的步骤在 VC# 2010 中创建示例:

(1) 可以修改 `FirstLINQQuery` 示例, 或者在 `C:\BegVCSharp\Chapter23` 目录中创建一个新的控制台应用程序 `23-2-LINQMethodSyntax`。打开主源文件 `Program.cs`。

(2) Visual C# 2010 会自动在 `Program.cs` 中包含 `Linq` 名称空间:

```
using System.Linq;
```

(3) 在 `Program.cs` 的 `Main()` 方法中添加如下代码:



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

    var queryResults = names.Where(n => n.StartsWith("S"));

    Console.WriteLine("Names beginning with S: ");

    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

代码段 23-2-LINQMethodSyntax\Program.cs



(4) 编译并执行程序(可以按下 F5 键)。结果也是以 S 开头的 `names` 列表, 且按照它们在数组中声明的顺序排列, 如下所示:

```
Names beginning with S:
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:
```

### 示例的说明

与前面一样, Visual C# 2010 会自动引用 `System.Linq` 名称空间:

```
using System.Linq;
```

再次声明和初始化 `names` 数组, 创建相同的源数据:

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
    "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah"};
```

LINQ 查询是不同的, 它现在是 `Where()` 方法的调用, 而不是查询表达式:

```
var queryResults = names.Where(n => n.StartsWith("S"));
```

C#编译器把 Lambda 表达式 `n => n.StartsWith("S")` 编译为一个匿名方法, `Where()` 在 `names` 数组的每个元素上执行这个方法。如果 Lambda 表达式给某个元素返回 `true`, 该元素就包含在 `Where()` 返回的结果集中。C#编译器从输入数据源(这里是 `names` 数组)的定义中推断, 该 `Where()` 方法应把 `string` 作为每个元素的输入类型。

许多工作都是在一行代码中完成的。对于像这样最简单的查询, 方法语法要比查询语法更加简短, 因为不需要 `from` 或 `select` 子句, 但是, 大多数查询都比这更复杂。

示例的剩余部分与前面的代码相同——在 `foreach` 循环中显示查询的结果, 并暂停输出, 以便在程序结束执行前看到结果:

```
foreach (var item in queryResults) {
    Console.WriteLine(item);
}

Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

这里不重复说明这些代码行, 因为本章第一个示例后面的“示例的说明”已经解释过了。下面继续研究如何使用 LINQ 的更多功能。

## 23.3 排序查询结果

用 `where` 子句(或者 `Where()` 方法调用)找到了感兴趣的数据后, LINQ 还可以方便地对得到的数据执行进一步处理, 例如, 重新排列结果的顺序。下面的示例将以字母顺序给第一个查询的结果

排序。

### 试一试：给查询结果排序

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 可以修改 FirstLINQQuery 示例，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序项目 23-3-OrderQueryResults。

(2) 打开主源文件 Program.cs。与以前一样，Visual C# 2010 会自动在 Program.cs 中包含 System.Linq 名称空间。

(3) 在 Program.cs 的 Main()方法中添加如下代码：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

    var queryResults =
        from n in names
        where n.StartsWith("S")
        orderby n
        select n;

    Console.WriteLine("Names beginning with S ordered alphabetically:");

    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

代码段 23-3-OrderQueryResults\Program.cs

(4) 编译并执行程序。结果是以 S 开头的 names 列表，且按字母顺序排序，如下所示：

```
Names beginning with S:
Samba
Singh
Small
Smith
Smythe
Program finished, press Enter/Return to continue:
```

#### 示例的说明

这个程序与前一个程序几乎相同，只是在查询语句中增加了一行代码：

```
var queryResults =
    from n in names
    where n.StartsWith("S")
    orderby n
    select n;
```

## 23.4 orderby 子句

orderby 子句如下所示：

```
orderby n
```

与 where 子句一样，orderby 子句也是可选的。只要添加一行，就可以给任意查询的结果排序，如果不使用 LINQ，就需要添加至少几行代码和几个方法或集合，来存储根据选择实现的排序算法重新排序的结果。如果有多个需要排序的类型，就需要为每个类型实现一系列排序方法。而使用 LINQ 不需要做这些工作，只需在查询语句中添加一个子句即可。

orderby 子句默认为升序(A 到 Z)，但可以添加 descending 关键字，以便指定降序(Z 到 A)：

```
orderby n descending
```

这会使示例的结果变成：

```
Smythe
Smith
Small
Singh
Samba
```

另外，可以按照任意表达式进行排序，而无需重新编写查询。例如，要按照姓名中的最后一个字母排序，而不是按一般的字母顺序排序，就只需添加如下 orderby 子句：

```
orderby n.Substring(n.Length - 1)
```

结果如下：

```
Samba
Smythe
Smith
Singh
Small
```



最后一个字母按字母顺序排序(a, e, h, h, l)。但这个执行过程依赖于实现方式。即无法保证除了 orderby 子句中指定的内容之外的其他字母的顺序。由于仅考虑最后一个字母，所以在本例中，Smith 在 Singh 的前面。

## 23.5 用方法语法排序

要给使用方法语法的查询添加排序等功能，只需给每个要在基于方法的 LINQ 查询上执行的 LINQ 操作添加一个方法调用，这也很简单。

试一试：使用方法语法排序

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 可以修改 23-2-LINQMethodSyntax 示例，或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序项目 23-4-OrderMethodSyntax。

(2) 在 Program.cs 的 Main()方法中添加如下代码，在所有的示例中，Visual C# 2010 都会自动包含对 System.Linq 名称空间的引用。



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

    var queryResults = names.OrderBy(n => n).Where(n => n.StartsWith("S"));

    Console.WriteLine("Names beginning with S: ");
    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue: ");
    Console.ReadLine();
}
```

代码段 23-4-OrderMethodSyntax\Program.cs

(3) 编译并执行程序。结果是以 S 开头的 names 列表，且按字母顺序排序，与前一个示例的输出结果相同。

示例的说明

这个程序与前一个方法语法示例几乎相同，只是在 Where()方法调用前面添加了 LINQ 方法 OrderBy()的调用：

```
var queryResults = names.OrderBy(n => n).Where(n => n.StartsWith("S"));
```

键入代码时，会在 IntelliSense 中看到，OrderBy()方法返回一个 IOrderedEnumerable，这是 IEnumerable 接口的一个超集，所以可以在其上调用 Where()方法，与其他返回 IEnumerable 接口的方法一样。



编译器推断，目前处理的是 string 数据，所以在 IntelliSense 中显示的数据类型是 IOrderedEnumerable<string>和 IEnumerable<string>。

需要把一个 Lambda 表达式传送给 OrderBy()方法，告诉它用于排序的函数是什么。我们传送了最简单的 Lambda 表达式 n=>n，因为只需要按照元素本身排序。在查询语法中，不需要创建这个额外的 Lambda 表达式。

为了给元素逆序排序，可以调用 OrderByDescending()方法：

```
var queryResults = names.OrderByDescending(n => n).Where(n => n.StartsWith("S"));
```

这会生成与查询语法版本中使用的 `orderby n descending` 子句相同的结果。

如果不按照元素的值排序，可以修改传送给 `OrderBy()` 方法的 Lambda 表达式。例如，要按照每个姓名的最后一个字母排序，可以使用 Lambda 表达式 `n => n.Substring(n.Length-1)`，把它传送给 `OrderBy` 方法，如下所示：

```
var queryResults =
    names.OrderBy(n => n.Substring(n.Length-1)).Where(n => n.StartsWith("S"));
```

这会生成与上一个示例相同的结果，但按照每个姓名的最后一个字母排序。

## 23.6 查询大型数据集

这个 LINQ 语法非常好，但其要点是什么？我们只要查看源数组，就可以看出需要的结果，为什么要查询这种一眼就能看出结果的数据源呢？如前所述，有时查询的结果不那么明显。在下面的示例中，就创建了一个非常大的数字数组，并用 LINQ 查询它。

### 试一试：查询大型数据集

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 `C:\BegVCSharp\Chapter23` 目录中创建一个新的控制台应用程序 `23-5-LargeNumberQuery`。与以前一样，创建项目时，Visual C# 2010 会自动在 `Program.cs` 中包含 `Linq` 名称空间。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

(2) 在 `Main()` 方法中添加如下代码：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    int[] numbers = generateLotsOfNumbers(12345678);

    var queryResults =
        from n in numbers
        where n < 1000
        select n
        ;

    Console.WriteLine("Numbers less than 1000: ");
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }
    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

代码段 23-5-LargeNumberQuery\Program.cs

(3) 添加如下方法，生成一个随机数列表：

```
private static int[] generateLotsOfNumbers(int count)
{
    Random generator = new Random(0);
    int[] result = new int[count];
    for (int i = 0; i < count; i++)
    {
        result[i] = generator.Next();
    }
    return result;
}
```

(4) 编译并执行程序。结果是一个小于 1000 的数字列表，如下所示：

```
Numbers less than 1000:
714
24
677
350
257
719
584
Program finished, press Enter/Return to continue:
```

### 示例的说明

与前面一样，第一步是引用 `System.Linq` 名称空间，这是在创建项目时由 Visual C# 2010 自动引用的：

```
using System.Linq;
```

接着创建一些数据，本例中是创建并调用 `generateLotsOfNumbers()` 方法：

```
int[] numbers = generateLotsOfNumbers(12345678);

private static int[] generateLotsOfNumbers(int count)
{
    Random generator = new Random(0);
    int[] result = new int[count];
    for (int i = 0; i < count; i++)
    {
        result[i] = generator.Next();
    }
    return result;
}
```

这不是一个小数据集，数组中有 120 万个数字！在本章最后的一个练习中，需要修改传送给 `generateLotsOfNumbers()` 方法的 `size` 参数，生成数量不同的随机数，看看这会对查询结果有什么影响。在做练习时会看到，这里的 `size` 参数 12 345 678 非常大，足以生成一些小于 1 000 的随机数，从而获得为第一个查询显示的结果。

数值应随机分布在有符号的整数范围内(从 0 到超过 20 亿)。用种子值 0 创建随机数生成器，可以确保每次创建相同的随机数集合，这是可以重复的，所以会获得与此处相同的查询结果，但在尝

试一些查询之前，并不知道查询结果是什么。而 LINQ 使这些查询很容易编写。

查询语句本身类似于前面用于 `names` 数组的查询，也是选择某些满足条件的数字(这里是条件的数字小于 1 000):

```
var queryResults =
    from n in numbers
    where n < 1000
    select n
```

这次不需要 `orderby` 子句，但处理时间略长(对于这个查询，处理时间的变化不太明显，但下一个示例会改变选择条件，处理时间的变化就比较明显了)。

用 `foreach` 语句输出查询的结果，与前面的示例相同:

```
Console.WriteLine("Numbers less than 1000:");

foreach (var item in queryResults) {
    Console.WriteLine(item);
}
```

同样，将结果输出到控制台上，并读取一个字符以暂停输出:

```
Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
```

后面所有的示例都有暂停代码，但不再列出，因为每个示例的暂停代码都相同。

使用 LINQ，可以很容易地修改查询条件，以便演示数据集的不同特性。但是，根据查询返回的结果数，每次都输出所有的结果是没有意义的。下一节将说明 LINQ 提供的聚合运算符是如何处理这种情况的。

## 23.7 聚合运算符

查询给出的结果常常超出了我们的期望。如果要修改大数查询程序的条件，只需列出大于 1 000 的数字，而不是小于 1 000 的数字，这会得到非常多的查询结果，数字会不停地显示出来。

LINQ 提供了一组聚合运算符，可用于分析查询的结果，而无需迭代所有结果。表 23-1 中列出的聚合运算符是数字结果集最常用的运算符，例如，大数查询的结果就常用这些聚合运算符，如果读者使用过数据库查询语言如 SQL，就会很熟悉这些运算符。

表 23-1

运 算 符	说 明
Count()	结果的个数
Min()	结果中的最小值
Max()	结果中的最大值
Average()	数字结果的平均值
Sum()	所有数字结果的总和



还有更多的聚合运算符，如 `Aggregate()`，它们可以执行代码，并允许编写自己的聚合函数。但是，这些都用于高级用户，超出了本书的讨论范围。



聚合运算符返回一个简单的标量类型，而不是一系列结果，所以使用它们会强制立即执行查询，而不是延迟执行。

下面的示例修改大数查询，并使用聚合运算符和 LINQ 分析大数查询的大于版本中的结果集。

#### 试一试：数字聚合运算符

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 这个示例可以修改前面的 `LargeNumberQuery` 示例，或者在 `C:\BegVCSharp\Chapter23` 目录中创建一个新的控制台项目 `23-6-NumericAggregates`。

(2) 与以前一样，创建项目时，Visual C# 2010 会自动在 `Program.cs` 中包含 `Linq` 名称空间。只需修改 `Main()` 方法，如下面的代码和本示例其余部分所示。与上一个例子一样，这个查询也不使用 `orderby` 子句。但是 `where` 子句中的条件与前一个例子相反(数字应大于 1 000( $n > 1000$ )，而不是小于 1 000)。



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    int[] numbers = generateLotsOfNumbers(12345678);

    Console.WriteLine("Numeric Aggregates");

    var queryResults =
        from n in numbers
        where n > 1000
        select n
        ;

    Console.WriteLine("Count of Numbers > 1000");
    Console.WriteLine(queryResults.Count());

    Console.WriteLine("Max of Numbers > 1000");
    Console.WriteLine(queryResults.Max());

    Console.WriteLine("Min of Numbers > 1000");
    Console.WriteLine(queryResults.Min());

    Console.WriteLine("Average of Numbers > 1000");
    Console.WriteLine(queryResults.Average());

    Console.WriteLine("Sum of Numbers > 1000");
    Console.WriteLine(queryResults.Sum(n => (long) n));

    Console.Write("Program finished, press Enter/Return to continue: ");
    Console.ReadLine();
}
```

代码段 23-6-NumericAggregates\Program.cs

(3) 添加上一个示例中使用的 `generateLotsOfNumbers()` 方法(如果不存在):

```
private static int[] generateLotsOfNumbers(int count)
{
    Random generator = new Random(0);
    int[] result = new int[count];
    for (int i = 0; i < count; i++)
    {
        result[i] = generator.Next();
    }
    return result;
}
```

(4) 编译并执行, 显示个数、最小值、最大值和平均值, 如下所示:

```
Numeric Aggregates
Count of Numbers > 1000
12345671
Maximum of Numbers > 1000
2147483591
Minimum of Numbers > 1000
1034
Average of Numbers > 1000
1073643807.50298
Sum of Numbers > 1000
13254853218619179
Program finished, press Enter/Return to continue:
```

这个查询生成的结果数量超过上一个例子(超过 120 万)。在这个结果集上使用 `orderby`, 对性能会有很显著的影响。结果集中的最大值超过 20 亿, 最小值刚刚大于 1 000。平均值大约是 10 亿, 接近数字范围的中间值。看来, `Rand()` 函数可以生成均匀分布的数字。

#### 示例的说明

程序的第一部分与上一个例子完全相同, 也是引用 `System.Linq` 名称空间, 然后用 `generateLotsOfNumbers()` 方法生成源数据:

```
int[] numbers = generateLotsOfNumbers(12345678);
```

查询也与上一个例子相同, 只是把 `where` 条件从小于改为大于:

```
var queryResults =
    from n in numbers
    where n > 1000
    select n;
```

如前所述, 使用大于条件的这个查询生成的结果远远多于小于查询(对这个数据集而言)。使用聚合运算符可以分析查询结果, 而无需输出每个结果, 或者在 `foreach` 循环中比较它们。每个聚合运算符都类似于一个可以在结果集上调用的方法, 也类似于在集合类型上调用的方法。

下面看看聚合运算符的用法:

- `Count()` 返回查询结果中的行数, 在这个例子中是 12 345 671 行:

```
Console.WriteLine("Count of Numbers > 1000");
```

```
Console.WriteLine(queryResults.Count());
```

- **Max()** 返回查询结果中的最大值，在这个例子中是大于 20 亿的一个数 2 147 483 591，它非常接近 `int` 的最大值(`int.MaxValue` 或 2 147 483 647)。

```
Console.WriteLine("Max of Numbers > 1000");
Console.WriteLine(queryResults.Max());
```

- **Min()** 返回查询结果中的最小值，在这个例子中是 1 034。

```
Console.WriteLine("Min of Numbers > 1000");
Console.WriteLine(queryResults.Min());
```

- **Average()** 返回查询结果中的平均值，在这个例子中是 1 073 643 807.502 98，它非常接近 1 000 到 20 亿的值范围的中间值。对于随机的大数而言，这个中间值没有什么意义，但说明了可以对查询结果进行分析。本章最后一部分将使用这些运算符对面向业务的数据进行更切合实际的分析。

```
Console.WriteLine("Average of Numbers > 1000");
Console.WriteLine(queryResults.Average());
```

- **Sum()**，在此给 `Sum()` 方法调用传送了 Lambda 表达式 `n=>(long) n`，以获得所有数字的总和。与 `Count()`、`Min()`、`Max()` 等相同，`Sum()` 有一个无参数的重载版本，但使用 `Sum()` 方法的这个版本会导致溢出错误，因为数据集中的数字太多，它们的总和太大，不能放在 `Sum()` 方法的无参数重载版本返回的标准的 32 位 `int` 中。Lambda 表达式允许把 `Sum()` 方法的结果转换为 64 位长整数，它可以保存超过  $13^{10}$  的数字 13 254 853 218 619 179，而不出现溢出。Lambda 表达式允许执行这个转换。

```
Console.WriteLine("Sum of Numbers > 1000");
Console.WriteLine(queryResults.Sum(n => (long) n));
```



**Count()** 返回 32 位 `int`，LINQ 还提供了一个 `LongCount()` 方法，它在 64 位整数中返回查询结果的个数。但有一个特殊情况：如果需要数字的 64 位版本，所有其他运算符都需要一个  $\lambda$  表达式或转换方法调用。

## 23.8 查询复杂的对象

前面的例子说明了 LINQ 查询可以处理各种简单类型，例如，数字和字符串。下面看看如何使用 LINQ 查询较复杂的对象。我们要创建一个简单的 `Customer` 类，它拥有的信息足以创建一些有趣的查询。

**试一试：查询复杂的对象**

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-7-QueryComplex Objects。

(2) 在 Program.cs 的 Program 类开头, 给 Customer 类添加如下的简短类定义:



```
class Customer
{
    public string ID { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public string Region { get; set; }
    public decimal Sales { get; set; }

    public override string ToString()
    {
        return "ID: " + ID + "City:" + City + "Country:" + Country +
            "Region: " + Region + "Sales: " + Sales;
    }
}
```

代码段 23-7-QueryComplexObjects\Program.cs

(3) 在 Program.cs 的 Program 类的 Main()方法中添加如下代码:



```
static void Main(string[] args)
{
    List < Customer > customers = new List < Customer > {
        new Customer { ID="A", City="New York", Country="USA",
            Region="North America", Sales=9999 },
        new Customer { ID="B", City="Mumbai", Country="India",
            Region="Asia", Sales=8888 },
        new Customer { ID="C", City="Karachi", Country="Pakistan",
            Region="Asia", Sales=7777 },
        new Customer { ID="D", City="Delhi", Country="India",
            Region="Asia", Sales=6666 },
        new Customer { ID="E", City="S o Paulo", Country="Brazil",
            Region="South America", Sales=5555 },
        new Customer { ID="F", City="Moscow", Country="Russia",
            Region="Europe", Sales=4444 },
        new Customer { ID="G", City="Seoul", Country="Korea", Region="Asia",
            Sales=3333 },
        new Customer { ID="H", City="Istanbul", Country="Turkey",
            Region="Asia", Sales=2222 },
        new Customer { ID="I", City="Shanghai", Country="China", Region="Asia",
            Sales=1111 },
        new Customer { ID="J", City="Lagos", Country="Nigeria",
            Region="Africa", Sales=1000 },
        new Customer { ID="K", City="Mexico City", Country="Mexico",
            Region="North America", Sales=2000 },
        new Customer { ID="L", City="Jakarta", Country="Indonesia",
            Region="Asia", Sales=3000 },
        new Customer { ID="M", City="Tokyo", Country="Japan",
            Region="Asia", Sales=4000 },
        new Customer { ID="N", City="Los Angeles", Country="USA",
```

```

                Region="North America", Sales=5000 },
new Customer { ID="O", City="Cairo", Country="Egypt",
                Region="Africa", Sales=6000 },
new Customer { ID="P", City="Tehran", Country="Iran",
                Region="Asia", Sales=7000 },
new Customer { ID="Q", City="London", Country="UK",
                Region="Europe", Sales=8000 },
new Customer { ID="R", City="Beijing", Country="China",
                Region="Asia", Sales=9000 },
new Customer { ID="S", City="Bogotá", Country="Colombia",
                Region="South America", Sales=1001 },
new Customer { ID="T", City="Lima", Country="Peru",
                Region="South America", Sales=2002 }
};
var queryResults =
    from c in customers
    where c.Region == "Asia"
    select c
;
Console.WriteLine("Customers in Asia: ");
foreach (Customer c in queryResults)
{
    Console.WriteLine(c);
}
Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
}
}

```

---

代码段 23-7-QueryComplexObjects\Program.cs

---

#### (4) 编译并执行程序，结果是来自亚洲的顾客列表：

```

Customers in Asia:
ID: B City: Mumbai Country: India Region: Asia Sales: 8888
ID: C City: Karachi Country: Pakistan Region: Asia Sales: 7777
ID: D City: Delhi Country: India Region: Asia Sales: 6666
ID: G City: Seoul Country: Korea Region: Asia Sales: 3333
ID: H City: Istanbul Country: Turkey Region: Asia Sales: 2222
ID: I City: Shanghai Country: China Region: Asia Sales: 1111
ID: L City: Jakarta Country: Indonesia Region: Asia Sales: 3000
ID: M City: Tokyo Country: Japan Region: Asia Sales: 4000
ID: P City: Tehran Country: Iran Region: Asia Sales: 7000
ID: R City: Beijing Country: China Region: Asia Sales: 9000
Program finished, press Enter/Return to continue:

```

#### 示例的说明

在 `Customer` 类的定义中使用了 C# 的自动属性功能，来声明 `Customer` 类的公共属性 (ID、City、Country、Region、Sales)，而不必为每个属性显式编写私有实例变量和 `get/set` 代码：

```

class Customer
{
    public string ID { get; set; }
}

```

```
public string City { get; set; }
```

```
...
```

为 `Customer` 类编写的唯一一个方法是 `ToString()` 方法的重写版本, 该方法为 `Customer` 实例提供字符串表示:

```
public override string ToString()
{
    return "ID:" + ID + "City:" + City + "Country:" + Country +
        "Region:" + Region + "Sales:" + Sales;
}
```

使用这个 `ToString()` 方法可以简化查询结果的输出, 如后面的代码所示。

在 `Program` 类的 `Main()` 方法中, 用集合/对象初始化语法创建了一个强类型化的集合, 其类型是 `Customer`, 这样就不需要编写构造方法, 再调用该构造方法创建每个列表成员了:

```
List < Customer > customers = new List < Customer > {
    new Customer { ID="A", City="New York", Country="USA",
                  Region="North America", Sales=9999},
    new Customer { ID="B", City="Mumbai", Country="India",
                  Region="Asia", Sales=8888 },
    ...
}
```

顾客遍及世界各地, 我们的数据中有足够的地理信息, 可以为查询建立有趣的选择条件和组合条件。

在 `Main()` 方法中, 创建查询语句, 本例选择来自亚洲的顾客:

```
var queryResults =
    from c in customers
    where c.Region == "Asia"
    select c
;
```

这个查询应很眼熟, 在其他示例中使用的也是 `from...where...select` 查询, 只是结果列表中的每一项都是一个完整对象(`Customer`), 而不是简单的 `string` 或 `int`。接着在 `foreach` 循环中输出结果:

```
Console.WriteLine("Customers in Asia: ");
foreach (Customer c in queryResults)
{
    Console.WriteLine(c);
}
```

这个 `foreach` 循环不同于前面的示例。因为我们知道要查询 `Customer` 对象, 所以把迭代变量显式地声明为 `Customer` 类型:

```
foreach (Customer c in queryResults)
```

可以用关键字 `var` 声明 `c`, 编译器就会推断出这个迭代变量的类型应是 `Customer`, 但显式声明它会使代码更容易理解。

在循环中, 仅编写了一条语句:

```
{
```

```

        Console.WriteLine(c);
    }

```

这里没有显式输出 `Customer` 的字段，因为给 `Customer` 类添加了 `ToString()` 的重写版本。如果未提供这个 `ToString()` 重写版本，默认的 `ToString()` 方法仅输出类型名称，如下所示：

```

Customers in Asia:
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
Program finished, press Enter/Return to continue:

```

这并不是我们想要的结果。当然，总是可以显式输出令人感兴趣的 `Customer` 属性：

```

Console.WriteLine("Customer {0}: {1}, {2}", c.ID, c.City, c.Country);

```

但是，如果只对对象的几个属性感兴趣，把整个对象都放在查询中就很低效了。LINQ 很容易通过投影(projection)创建只包含所需条目的查询结果，如下一节所述。

## 23.9 投影：在查询中创建新对象

投影(projection)是在 LINQ 查询中从其他数据类型中创建新数据类型的技术术语。`select` 关键字是投影运算符，前面的示例中就使用了这个关键字。如果熟悉 SQL 数据查询语言中的 `SELECT` 关键字，就很熟悉从数据对象中选择某个字段的操作，而不是选择整个对象。在 LINQ 中，也可以这么做，例如，前面的例子只从 `Customer` 列表中选择 `City` 字段，只需修改查询语句中的 `select` 子句，以便仅引用 `City` 属性：

```

var queryResults =
    from c in customers
    where c.Region == "Asia"
    select c.City
;

```

这会生成如下结果：

```

Mumbai
Karachi
Delhi
Seoul
Istanbul
Shanghai
Jakarta
Tokyo
Tehran

```



Beijing

甚至可以通过给 `select` 添加表达式, 来转换查询中的数据, 对于数字数据类型应如下所示:

```
select n + 1
```

对于字符串数据类型查询, 应如下所示:

```
select s.ToUpper()
```

但是, 与 SQL 不同, LINQ 不允许在 `select` 子句中有多个字段。这表示

```
select c.City, c.Country, c.Sales
```

会生成一个编译错误(需要分号), 因为 `select` 子句在其参数列表中只有一项。

在 LINQ 中, 应在 `select` 子句中动态创建一个新对象, 来保存查询的结果。如下面的示例所示。

### 试一试: 投射——在查询中创建新对象

按照下面的步骤在 Visual C# 2010 中创建示例:

(1) 修改 23-7-QueryComplexObjects, 或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-8-ProjectionCreateNewObjects。

(2) 如果选择创建新项目, 就从 23-7-QueryComplexObjects 示例中复制创建 `Customer` 类的代码和初始化顾客列表(`List<Customer> customers`)的代码, 这些代码与前面的代码相同。

(3) 在 `Main()` 方法中的 `customers` 列表初始化之后, 输入(或修改)查询, 使处理循环如下所示:



可从  
wrox.com  
下载源代码

```
var queryResults =
    from c in customers
    where c.Region == "North America"
    select new { c.City, c.Country, c.Sales };
foreach (var item in queryResults)
{
    Console.WriteLine(item);
}
```

代码段 23-8-ProjectionCreateNewObjects\Program.cs

(4) `Main()` 方法的其余代码与前面示例的相同。

(5) 编译并执行程序, 将列出来自北美的顾客的所选字段, 如下所示:

```
{ City = New York, Country = USA, Sales = 9999 }
{ City = Mexico City, Country = Mexico, Sales = 2000 }
{ City = Los Angeles, Country = USA, Sales = 5000 }
Program finished, press Enter/Return to continue:
```

### 示例的说明

`Customer` 类和 `customers` 列表的初始化与前面例子中的相同。在查询中, 把请求的区域改为北美, 会使事情复杂一些。对于投射, 有趣的改变在于 `select` 子句的参数:

```
select new { c.City, c.Country, c.Sales }
```

这里在 `select` 子句中直接使用 C# 匿名类型创建语法, 创建一个未命名的对象类型, 它带有 `City`、`Country` 和 `Sales` 属性。`select` 子句创建了新对象。这样, 只会复制这3个属性, 完成处理查询的不同阶段。

输出查询结果时, 使用了通用的 `foreach` 循环代码, 在前面的示例中也使用了相同的循环代码, 但 `Customers` 查询例外:

```
foreach (var item in queryResults)
{
    Console.WriteLine(item);
}
```

这段代码非常通用, 编译器会推断出查询结果的类型, 给匿名类型调用正确的方法, 我们无需显式编写代码。甚至不需要提供 `ToString()` 重写方法, 因为编译器提供了 `ToString()` 方法的默认实现代码, 以类似于对象初始化的方式输出属性名称和值。

## 23.10 投影: 方法语法

投影查询的方法语法版本是通过把 LINQ 方法 `Select()` 的调用关联到我们调用的其他 LINQ 方法上来实现的。例如, 如果在 `Where()` 方法调用上添加 `Select()` 方法调用, 就会得到相同的查询结果:

```
var queryResults = customers.Where(c => c.Region == "North America")
                              .Select(c => new { c.City, c.Country, c.Sales });
```

在查询语法中需要 `select` 子句, 但在以前并没有看到 `Select()` 方法, 因为在 LINQ 方法语法中不需要它, 除非在进行投影(改变结果集中所查询的原始类型)。

方法调用顺序不固定, 因为 LINQ 方法调用返回的类型执行了 `IEnumerable`——可以在 `Where()` 方法的结果上调用 `Select()` 方法, 反之亦然。但是, 根据查询的特性, 调用顺序也许很重要。例如, 不能像下面这样颠倒 `Select()` 和 `Where()` 的顺序:

```
var queryResults = customers.Select(c => new { c.City, c.Country, c.Sales })
                              .Where(c => c.Region == "North America");
```

`Region` 属性未包含在 `Select()` 投影创建的匿名类型(`c.City`, `c.Country`, `c.Sales`)中, 所以程序会在 `Where()` 方法上得到一个编译错误, 表示匿名类型不包含 `Region` 的定义。

但是, 如果 `Where()` 方法根据包含在匿名类型中的字段(如 `City`)来限制数据, 就不会有问题——例如, 下面的查询会正常编译并执行:

```
var queryResults = customers.Select(c => new { c.City, c.Country, c.Sales })
                              .Where(c => c.City == "New York");
```

## 23.11 单值选择查询

在 SQL 数据查询语言中, 我们熟悉的另一类查询是 `SELECT DISTINCT` 查询, 该查询可搜索数据中的唯一值, 也就是说, 值是不重复的。这是使用查询时的一个常见需求。

假定需要在前面示例使用的顾客数据中查找不同的区域，由于在这些数据中没有单独的区域列表，所以需要从顾客列表中找到唯一的、不重复的区域列表。LINQ 提供了 `Distinct()` 方法，以便找出这些数据，如下面的示例所示。

### 试一试：投影——单值选择查询

按照下面的步骤在 Visual C# 2010 中创建示例：

(1) 修改前面的示例 23-8-ProjectionCreateNewObjects，或者在 `C:\BegVCSharp\Chapter23` 目录中创建一个新的控制台应用程序 23-9-SelectDistinctQuery。

(2) 从 23-7-QueryComplexObjects 示例中复制创建 `Customer` 类的代码和初始化顾客列表 (`List<Customer> customers`) 的代码，这些代码与前面的代码相同。

(3) 在 `Main()` 方法的 `customers` 列表初始化之后，输入(或修改)查询，如下所示：



可从  
wrox.com  
下载源代码

```
var queryResults = customers.Select(c => c.Region).Distinct();
```

代码段 23-9-SelectDistinctQuery\Program.cs

(4) `Main()` 方法的其余代码与前面例子的相同。

(5) 编译并执行程序，结果显示的是顾客所在的唯一区域，如下所示：

```
North America
Asia
South America
Europe
Africa
Program finished, press Enter/Return to continue:
```

#### 示例的说明

`Customer` 类和 `customers` 列表的初始化与前面例子中的相同。在查询语句中，调用了 `Select()` 方法，用一个简单的 `Lambda` 表达式从 `Customer` 对象中选择区域，再调用 `Distinct()` 从 `Select()` 中返回唯一的结果：

```
var queryResults = customers.Select(c => c.Region).Distinct();
```

只能在方法语法中使用 `Distinct()`，所以使用方法语法调用 `Select()`。还可以调用 `Distinct()` 来修改在查询语法中创建的查询：

```
var queryResults = (from c in customers select c.Region).Distinct();
```

查询语法由 C# 编译器转换为方法语法中的同系列 LINQ 方法调用，所以如果可以改进可读性和样式，可以混合和匹配它们。

## 23.12 Any 和 All

我们常常需要的另一类查询是确定数据是否满足某个条件，或者确保所有数据都满足某个条件。例如，需要确定某个产品是否已经脱销(库存为 0)，或者是否发生了某个交易。

LINQ 提供了两个布尔方法: Any()和 All(), 它们可以快速确定对于数据而言, 某个条件是 true 还是 false。因此很容易地找到数据, 如下面的示例所示。

试一试: 使用 Any()和 All()

- 按照下面的步骤在 Visual C# 2010 中创建示例:
- (1) 修改前面的示例 23-9-SelectDistinctQuery, 或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-10-AnyAndAll。
  - (2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码, 这些代码与前面的代码相同。
  - (3) 在 Main()方法中, 在 customers 列表初始化和查询声明后, 删除处理循环, 输入如下所示的代码:



可从  
wrox.com  
下载源代码

```
bool anyUSA = customers.Any(c => c.Country == "USA");
if (anyUSA)
{
    Console.WriteLine("Some customers are in the USA");
}
else
{
    Console.WriteLine("No customers are in the USA");
}

bool allAsia = customers.All(c => c.Region == "Asia");
if (allAsia)
{
    Console.WriteLine("All customers are in Asia");
}
else
{
    Console.WriteLine("Not all customers are in Asia");
}
```

代码段 23-10-AnyAndAll\Program.cs

- (4) Main()方法的其余代码与前面例子的相同。
- (5) 编译并执行程序, 将看到一些消息, 指出一些顾客来自美国, 并不是所有的顾客都来自亚洲:

```
Some customers are in the USA
Not all customers are in Asia
Program finished, press Enter/Return to continue:
```

**示例的说明**

Customer 类和 customers 列表的初始化与前面例子中的相同。在第一个查询语句中, 调用了 Any()方法, 用一个简单的 Lambda 表达式检查 Customer Country 字段的值是否是 USA:

```
bool anyUSA = customers.Any(c => c.Country == "USA");
```

LINQ 方法 Any()把传送给它的 Lambda 表达式 c=>c.Country == "USA"应用于 customers 列表中

的所有数据, 如果对于列表中的任意顾客, Lambda 表达式是 true, 就返回 true。

接着, 检查 Any() 方法返回的布尔结果变量, 输出一个消息, 显示查询的结果(Any() 方法虽然仅返回 true 或 false, 但它会执行一个查询, 得到 true 或 false 结果):

```
if (anyUSA)
{
    Console.WriteLine("Some customers are in the USA");
}
else
{
    Console.WriteLine("No customers are in the USA");
}
```

虽然可以通过一些巧妙的代码使这个消息更紧凑一些, 但这里的代码比较直观, 容易理解。anyUSA 变量设置为 true, 因为数据集中的确有顾客居住在美国, 所以看到了消息 Some customers are in the USA。

在下一个查询语句中, 调用了 All() 方法, 利用另一个简单的 Lambda 表达式确定是否所有的顾客都来自亚洲:

```
bool allAsia = customers.All(c => c.Region == "Asia");
```

LINQ 方法 All() 把 Lambda 表达式应用于数据集, 并返回 false, 因为有一些顾客不是来自亚洲。然后根据 allAsia 的值返回相应的消息。

## 23.13 多级排序

处理了带多个属性的对象后, 就要考虑另一种情形了: 按一个字段给查询结果排序是不够的, 需要查询顾客, 并按照区域使结果以字母顺序排列, 再按照区域中国家或城市名称来排序。使用 LINQ, 可以方便地完成这个任务, 如下面的示例所示。

### 试一试: 多级排序

按照下面的步骤在 Visual C# 2010 中创建示例:

- (1) 修改前面的示例 23-8-ProjectionCreateNewObjects, 或者在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-11-MultiLevelOrdering。
- (2) 如 23-7-QueryComplexObjects 示例所示, 创建 Customer 类并初始化顾客列表(List<Customer> customers), 这些代码与前面示例中的代码完全相同。
- (3) 在 Main() 方法的 customers 列表初始化之后, 输入如下所示的查询:



可从  
wrox.com  
下载源代码

```
var queryResults =
    from c in customers
    orderby c.Region, c.Country, c.City
    select new { c.ID, c.Region, c.Country, c.City }
;
```

代码段 23-11-MultiLevelOrdering\Program.cs

(4) 结果处理循环和 `Main()` 方法的其余代码与前面例子中的相同。

(5) 编译并执行程序，从所有顾客中选择出来的属性将先按区域排序，再按国家排序，最后按城市排序，如下所示：

```
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = E, Region = South America, Country = Brazil, City = São Paulo }
{ ID = S, Region = South America, Country = Colombia, City = Bogotá }
{ ID = T, Region = South America, Country = Peru, City = Lima }
Program finished, press Enter/Return to continue:
```

#### 示例的说明

`Customer` 类和 `customers` 列表的初始化与前面例子的相同。因为要查看所有的顾客，这个查询中没有 `where` 子句，但按顺序列出了要排序的字段，它们放在 `orderby` 子句的一个用逗号分开的列表中：

```
orderby c.Region, c.Country, c.City
```

这很容易，但不太直观，这个简单的字段列表允许放在 `orderby` 子句中，但不能放在 `select` 子句中，这就是 LINQ 的工作方式。如果知道 `select` 子句会创建一个新对象，而根据定义，`orderby` 子句会逐个字段地执行，就不会觉得这个字段列表难以理解了。

可以给列出的任意字段添加 `descending` 关键字，反转该字段的排序顺序。例如，若要对查询结果要按照区域升序排序，再按照国家降序排序，只需在列表中的 `Country` 后面加上 `descending` 关键字即可，如下所示：

```
orderby c.Region, c.Country descending, c.City
```

添加了 `descending` 关键字后，结果如下：

```
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
```



```

{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = T, Region = South America, Country = Peru, City = Lima }
{ ID = S, Region = South America, Country = Colombia, City = Bogotá }
{ ID = E, Region = South America, Country = Brazil, City = São Paulo }
Program finished, press Enter/Return to continue:

```

注意，即使国家的顺序被反转了，印度和中国的城市仍按升序排序。

## 23.14 多级排序方法语法：ThenBy

使用方法语法进行多级排序时，后台的操作比较复杂，它使用了 `ThenBy()` 和 `OrderBy()` 方法。例如，下面的代码会得到与前面创建的示例相同的查询结果：

```

var queryResults = customers.OrderBy(c => c.Region)
    .ThenBy(c => c.Country)
    .ThenBy(c => c.City)
    .Select(c => new { c.ID, c.Region, c.Country, c.City });

```

多字段列表可以用在查询语法的 `orderby` 子句中，这是很明显的，因为它会转换为一系列 `ThenBy()` 方法调用，逐个字段地执行。编写这些方法调用的顺序非常重要，必须先编写 `OrderBy()`，因为 `ThenBy()` 方法只能在 `IOrderedEnumerable` 接口上使用，而 `IOrderedEnumerable` 接口是由 `OrderBy()` 生成的。但是，`ThenBy()` 方法可以根据需要多次(次数不限)关联到其他 `ThenBy()` 方法调用上，显然，查询语法比方法语法更容易编写。

如果第一个字段是以降序排序，就应调用 `OrderByDescending()` 来指定降序排序；如果其他字段要以降序排序，就应调用 `ThenByDescending()` 来指定。例如，在这个例子中，国家要以降序排列，方法语法的查询应如下所示：

```

var queryResults = customers.OrderBy(c => c.Region)
    .ThenByDescending(c => c.Country)
    .ThenBy(c => c.City)
    .Select(c => new { c.ID, c.Region, c.Country, c.City });

```

## 23.15 组合查询

组合查询(group query)把数据分解为组，允许按组来排序、计算聚合值以及进行比较。这常常是商务环境中最有趣的查询(它驱动了决策系统)。例如，要按照国家或区域比较销售量，确定在哪里开新店或雇佣更多的员工，如下面的示例所示。



试一试：组合查询

按照下面的步骤在 Visual C# 2010 中创建示例：

- (1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-12-GroupQuery。
- (2) 如 23-7-QueryComplexObjects 示例所示，创建 Customer 类并初始化顾客列表(List<Customer> customers)，这些代码与前面示例中的代码完全相同。
- (3) 在 Main()方法的 customers 列表初始化之后，输入如下所示的两个查询：



```
var queryResults =
    from c in customers
    group c by c.Region into cg
    select new { TotalSales = cg.Sum(c => c.Sales), Region = cg.Key }
;
var orderedResults =
    from cg in queryResults
    orderby cg.TotalSales descending
    select cg
;
```

代码段 23-12-GroupQuery\Program.cs

- (4) 在 Main()方法中，添加下面的输出语句和 foreach 处理循环：

```
Console.WriteLine("Total\t: By\nSales\t: Region\n-----\t -----");
foreach (var item in orderedResults)
{
    Console.WriteLine(item.TotalSales + "\t: " + item.Region);
}
```

- (5) 结果处理循环和Main()方法中的其余代码与前面例子中的相同。编译并执行程序，下面是组合结果：

```
Total : By
Sales : Region
-----
52997 : Asia
16999 : North America
12444 : Europe
8558 : South America
7000 : Africa
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子的相同。

组合查询中的数据通过一个键(key)字段来组合，每个组中的所有成员都共享这个字段值。在这个例子中，键字段是 Region:

```
group c by c.Region
```

要计算每个组的总和，应生成一个新的结果集 cg:

```
group c by c.Region into cg
```

在 `select` 子句中, 投影了一个新的匿名类型, 其属性是总销售量(通过引用 `cg` 结果集来计算)和组的键值, 后者是用特殊的组 `Key` 来引用的:

```
select new { TotalSales = cg.Sum(c => c.Sales), Region = cg.Key }
```

组的结果集实现了 LINQ 接口 `IGrouping`, 它支持 `Key` 属性。我们总是要以某种方式引用 `Key` 属性, 来处理组合的结果, 因为该属性表示创建数据中的每个组时使用的条件。

要按照 `TotalSales` 字段对结果降序排序, 以便查看某区域中的最高销售量、次高销售量等等, 需要创建第二个查询, 对组合查询的结果排序:

```
var orderedResults =
    from cg in queryResults
    orderby cg.TotalSales descending
    select cg
;
```

第二个查询是一个标准的 `select` 查询, 带一个 `orderby` 子句, 与前面示例中的相同。但它没有使用任何 LINQ 组合功能, 只是数据源来自于前面的组合查询。

接着输出结果, 用一些格式化代码显示带有列标题的数据, 在总销售量与组名之间显示了分隔符:

```
Console.WriteLine("Total\t: By\nSales\t: Region\n-----\t -----");
foreach (var item in orderedResults)
{
    Console.WriteLine(item.TotalSales + "\t: " + item.Region);
}
```

可以用更复杂的方式进行格式化, 指定字段宽度, 总销售量右对齐, 但这只是一个例子, 不需要这么多格式, 能看清数据, 理解代码做了些什么就足够了。

## 23.16 Take 和 Skip

假定需要数据集中销售量位于前 5 名的顾客。我们事先并不知道跻身前 5 名需要达到多高的销量, 所以不能使用 `where` 条件查找他们。

一些 SQL 数据库(如 Microsoft SQL Server)实现了 TOP 运算符, 所以可以执行命令 `SELECT Top 5 FROM ...`, 获得前 5 名顾客的数据。

与这个操作对应的 LINQ 方法是 `Take()`, 它可以从查询结果中提取前 `n` 个结果。实际上, 这个方法需要和 `orderby` 子句一起使用, 才能获得前 `n` 个结果。但 `orderby` 子句并不是必需的, 因为有时知道数据已经按指定的顺序排列好了, 或者只需要前 `n` 个结果, 而不必考虑它们的顺序。

`Take()`的反面是 `Skip()`, 它可以跳过前 `n` 个结果, 返回剩余的结果。`Take()`和 `Skip()`在 LINQ 文档说明中称为分区运算符(`partitioning operator`), 因为它们把结果集分为前 `n` 个结果(`Take()`)和/或其余的结果(`Skip()`)。

下面的示例对顾客列表数据使用了 `Take()`和 `Skip()`。

## 试一试：使用 Take 和 Skip

按照下面的步骤在 Visual C# 2010 中创建示例：

- (1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-13-TakeAndSkip。
- (2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表 (List<Customer> customers) 的代码。
- (3) 在 Main() 方法的 customers 列表初始化之后，输入如下所示的查询：



可从  
wrox.com  
下载源代码

```
//query syntax
var queryResults =
    from c in customers
    orderby c.Sales descending
    select new { c.ID, c.City, c.Country, c.Sales }
;
```

代码段 23-13-TakeAndSkip\Program.cs

- (4) 输入两个结果处理循环，一个使用 Take()，另一个使用 Skip()：

```
Console.WriteLine("Top Five Customers by Sales");
foreach (var item in queryResults.Take(5))
{
    Console.WriteLine(item);
}

Console.WriteLine("Customers Not In Top Five");
foreach (var item in queryResults.Skip(5))
{
    Console.WriteLine(item);
}
```

- (5) 编译并执行程序，结果显示的是前 5 名顾客和剩余的顾客：

```
Top Five Customers by Sales
{ ID = A, City = New York, Country = USA, Sales = 9999 }
{ ID = R, City = Beijing, Country = China, Sales = 9000 }
{ ID = B, City = Mumbai, Country = India, Sales = 8888 }
{ ID = Q, City = London, Country = UK, Sales = 8000 }
{ ID = C, City = Karachi, Country = Pakistan, Sales = 7777 }
Customers Not In Top Five
{ ID = P, City = Tehran, Country = Iran, Sales = 7000 }
{ ID = D, City = Delhi, Country = India, Sales = 6666 }
{ ID = O, City = Cairo, Country = Egypt, Sales = 6000 }
{ ID = E, City = Sao Paulo, Country = Brazil, Sales = 5555 }
{ ID = N, City = Los Angeles, Country = USA, Sales = 5000 }
{ ID = F, City = Moscow, Country = Russia, Sales = 4444 }
{ ID = M, City = Tokyo, Country = Japan, Sales = 4000 }
{ ID = G, City = Seoul, Country = Korea, Sales = 3333 }
{ ID = L, City = Jakarta, Country = Indonesia, Sales = 3000 }
{ ID = H, City = Istanbul, Country = Turkey, Sales = 2222 }
{ ID = T, City = Lima, Country = Peru, Sales = 2002 }
{ ID = K, City = Mexico City, Country = Mexico, Sales = 2000 }
{ ID = I, City = Shanghai, Country = China, Sales = 1111 }
```

```
{ ID = S, City = Bogotá, Country = Colombia, Sales = 1001 }
{ ID = J, City = Lagos, Country = Nigeria, Sales = 1000 }
Program finished, press Enter/Return to continue:
```

### 示例的说明

Customer 类和 customers 列表的初始化与前面例子中的相同。

主查询由查询语法的 from...orderby...select 语句组成, 类似于本章前面创建的查询, 只是其中没有 where 子句, 因为我们要获得所有的顾客(按销售量从高到低排序):

```
var queryResults =
    from c in customers
    orderby c.Sales descending
    select new { c.ID, c.City, c.Country, c.Sales }
```

这个示例与前面的示例略有不同: 这个示例在对查询结果执行 foreach 循环之前, 并没有应用运算符, 因为本例要重用查询结果。首先使用 Take(5) 获得前 5 名顾客:

```
foreach (var item in queryResults.Take(5))
```

接着使用 Skip(5) 跳过前 5 项(这些项刚才已经输出了), 从原来的查询结果集中输出剩余的顾客:

```
foreach (var item in queryResults.Skip(5))
```

输出结果和暂停屏幕显示的代码与前面示例中的相同, 只是消息有一点儿变化, 这里不再重复。

## 23.17 First 和 FirstOrDefault

假定需要在数据集中查找一名来自非洲的顾客, 我们需要实际的数据, 而不是 true/false 值或者包含所有匹配值的结果集。

LINQ 通过 First() 方法提供了这个功能, 它返回结果集中第一个匹配给定条件的元素。如果没有来自非洲的顾客, LINQ 还提供了方法 FirstOrDefault() 来处理这种情况, 而无需添加错误处理代码。

下面的示例给 customers 列表数据使用了 First() 和 FirstOrDefault() 方法。

试一试: 使用 First 和 FirstOrDefault

按照下面的步骤在 Visual C# 2010 中创建示例:

- (1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-14-FirstOrDefault。
- (2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表 (List<Customer>customers) 的代码。
- (3) 在 Main() 方法的 customers 列表初始化之后, 输入如下所示的查询:



可从  
wrox.com  
下载源代码

```
var queryResults = from c in customers
                    select new { c.City, c.Country, c.Region }
                    ;
```

代码段 23-14-FirstOrDefault\Program.cs

(4) 使用 First() 和 FirstOrDefault() 方法输入如下查询:

```
Console.WriteLine("A customer in Africa");
Console.WriteLine(queryResults.First(c => c.Region == "Africa"));

Console.WriteLine("A customer in Antarctica");
Console.WriteLine(queryResults.FirstOrDefault(c => c.Region == "Antarctica"));
```

(5) 编译并执行程序, 结果如下:

```
A customer in Africa
{ City = Lagos, Country = Nigeria, Region = Africa }
A customer in Antarctica

Program finished, press Enter/Return to continue:
```

#### 示例的说明

Customer 类和 customers 列表的初始化与前面示例中的相同。

主查询由查询语法的 from...orderby...select 语句组成, 类似于本章前面创建的查询, 只是其中没有 where 和 orderby 子句。我们用 select 语句投影了感兴趣的字段。本例中选择了 City、Country 和 Region 属性:

```
var queryResults = from c in customers
                    select new { c.City, c.Country, c.Region }
                    ;
```

First() 运算符返回一个对象值, 而不是结果集, 所以不需要创建 foreach 循环, 直接输出结果即可:

```
Console.WriteLine(queryResults.First(c => c.Region == "Africa"));
```

这行代码找到了一名顾客, 输出 City=Lagos, Country=Nigeria, Region=Africa 结果。接着使用 FirstOrDefault() 查询南极洲区域:

```
Console.WriteLine(queryResults.FirstOrDefault(c => c.Region == "Antarctica"));
```

这行代码找不到任何结果, 所以返回空(空结果集), 输出空白。如果给南极洲查询使用 First() 运算符, 而不是 FirstOrDefault(), 就会得到如下异常:

```
System.InvalidOperationException: Sequence contains no matching element
```

使用 FirstOrDefault() 时, 当查询条件不满足时, 将为 customers 列表返回默认元素, 而使用 First() 时则是返回 null(因为这些元素的类型未知)。因而这里对南极洲地区查询时, 您会收到一条异常消息。

输出结果和暂停屏幕显示的代码与前面示例中的相同, 只是消息有一点儿变化。

## 23.18 集运算符

LINQ 提供了标准的集运算符(set operator), 如 Union() 和 Intersect(), 对查询结果执行操作。在前面编写 Distinct() 查询时, 就使用了一个集运算符。

下面的示例添加了一个简单的订单列表，它是由假想的顾客提交的，并使用标准的集运算符来匹配已有的顾客。

### 试一试：集运算符

按照下面的步骤在 Visual C# 2010 中创建示例：

- (1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-15-SetOperators。
- (2) 从 23-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表 (List<Customer>customers) 的代码。
- (3) 在 Customer 类的后面添加如下 Order 类：



可从  
wrox.com  
下载源代码

```
class Order
{
    public string ID { get; set; }
    public decimal Amount { get; set; }
}
```

代码段 23-15-SetOperators\Program.cs

- (4) 在 Main() 方法的 customers 列表初始化之后，用如下数据创建并初始化一个 orders 列表：

```
List < Order > orders = new List < Order > {
    new Order { ID="P", Amount=100 },
    new Order { ID="Q", Amount=200 },
    new Order { ID="R", Amount=300 },
    new Order { ID="S", Amount=400 },
    new Order { ID="T", Amount=500 },
    new Order { ID="U", Amount=600 },
    new Order { ID="V", Amount=700 },
    new Order { ID="W", Amount=800 },
    new Order { ID="X", Amount=900 },
    new Order { ID="Y", Amount=1000 },
    new Order { ID="Z", Amount=1100 }
};
```

- (5) 在 orders 列表初始化后，输入如下查询：

```
var customerIDs =
    from c in customers
    select c.ID
;
var orderIDs =
    from o in orders
    select o.ID
;
```

- (6) 用 Intersect() 输入如下查询：

```
var customersWithOrders = customerIDs.Intersect(orderIDs);
Console.WriteLine("Customer IDs with Orders: ");
foreach (var item in customersWithOrders)
{
    Console.WriteLine("{0} ", item);
}
```

```

}
Console.WriteLine();

```

(7) 接着, 用 `Except()` 输入如下查询:

```

Console.WriteLine("Order IDs with no customers: ");
var ordersNoCustomers = orderIDs.Except(customerIDs);
foreach (var item in ordersNoCustomers)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();

```

(8) 最后, 用 `Union()` 输入如下查询:

```

Console.WriteLine("All Customer and Order IDs: ");
var allCustomerOrderIDs = orderIDs.Union(customerIDs);
foreach (var item in allCustomerOrderIDs)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();

```

(9) 编译并执行程序, 结果如下:

```

Customers IDs with Orders:
P Q R S T
Order IDs with no customers:
U V W X Y Z
All Customer and Order IDs:
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Program finished, press Enter/Return to continue:

```

### 示例的说明

`Customer` 类和 `customers` 列表的初始化与前面示例中的相同。新的 `Order` 类与 `Customer` 类是类似的, 也使用了 C# 自动属性功能来声明公共属性 (`ID`、`Amount`):

```

class Order
{
    public string ID { get; set; }
    public decimal Amount { get; set; }
}

```

与 `Customer` 类一样, 这也是一个简化的例子, 只包含足以使查询工作的数据。使用两个简单的 `from...select` 查询来获取 `Customer` 类和 `Order` 类的 `ID` 字段:

```

var customerIDs =
    from c in customers
    select c.ID
;
var orderIDs =
    from o in orders
    select o.ID
;

```



接着使用 `Intersect()` 运算符查找也在 `orderIDs` 结果中有订单的顾客 ID。只有在两个结果集中都有的 ID 才包含在交集中：

```
var customersWithOrders = customerIDs.Intersect(orderIDs);
```



运算符要求集成员有相同的类型，才能确保得到希望的结果。这里利用了一个事实：两个对象类型中的 ID 都是字符串，有相同的语义(类似于数据库中的外键)。

结果集的输出利用了一个事实：ID 是单个字符，所以使用 `Console.Write()` 调用，而不是 `WriteLine()` 调用，直到 `foreach` 循环的最后才使用 `WriteLine()`，使输出紧凑、简洁：

```
Console.WriteLine("Customer IDs with Orders: ");
foreach (var item in customersWithOrders)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();
```

在其余的 `foreach` 循环中也使用了这个输出逻辑。

接着使用 `Except()` 运算符查找没有匹配顾客的订单 ID：

```
Console.WriteLine("Order IDs with no customers: ");
var ordersNoCustomers = orderIDs.Except(customerIDs);
```

最后使用 `Union()` 运算符查找所有顾客 ID 和订单 ID 字段的并集：

```
Console.WriteLine("All Customer and Order IDs: ");
var allCustomerOrderIDs = orderIDs.Union(customerIDs);
```

注意，ID 的输出顺序与它们在顾客和订单列表中的顺序相同，只是删除了重复的项。

暂停屏幕显示的代码与前面示例中的相同。

运算符非常有用，但它们的实际用处因为受到所有对象的类型都必须相同的限制而缩小了。在需要处理类型类似的结果集等少数情况下，这些运算符是很有用的。但在更常见的需要处理不同相关对象类型的情况下，应采用专门为处理不同对象类型而设计的机制，例如，`join` 语句。

## 23.19 Join 查询

刚才用一个共享的键字段(ID)创建的 `customers` 和 `orders` 列表等数据集可以执行 `join` 查询，即可以用一个查询搜索两个列表中相关的数据，用键字段把结果连接起来。这类似于 SQL 数据查询语言中的 `JOIN` 操作。LINQ 在查询语法中提供了 `join` 命令，如下面的示例所示。

试一试: Join 查询

按照下面的步骤在 Visual C# 2010 中创建示例:

- (1) 在 C:\BegVCSharp\Chapter23 目录中创建一个新的控制台应用程序 23-16-JoinQuery。
- (2) 从前面的示例中复制创建 Customer 类和 Order 类的代码, 以及初始化顾客列表 (List<Customer>customers) 和订单列表(List<Order>orders)的代码。
- (3) 在 Main()方法的 customers 和 orders 列表初始化之后, 输入如下所示的查询:



```
var queryResults =
    from c in customers
    join o in orders on c.ID equals o.ID
    select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = o.Amount,
                SalesAfter = c.Sales+o.Amount };
```

代码段 23-16-JoinQuery\Program.cs

- (4) 用前面例子中使用的标准 foreach 查询处理循环结束程序:

```
foreach (var item in queryResults)
{
    Console.WriteLine(item);
}
```

- (5) 编译并执行程序, 结果如下:

```
{ ID = P, City = Tehran, SalesBefore = 7000, NewOrder = 100, SalesAfter = 7100 }
{ ID = Q, City = London, SalesBefore = 8000, NewOrder = 200, SalesAfter = 8200 }
{ ID = R, City = Beijing, SalesBefore = 9000, NewOrder = 300, SalesAfter = 9300 }
{ ID = S, City = Bogotá, SalesBefore = 1001, NewOrder = 400, SalesAfter = 1401 }
{ ID = T, City = Lima, SalesBefore = 2002, NewOrder = 500, SalesAfter = 2502 }
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 Order 类以及 customers 和 orders 列表的声明和初始化与前面示例中的相同。

查询使用 join 关键字通过 Customer 类和Order 类的 ID 字段, 把每个顾客与其对应的订单连接起来:

```
var queryResults =
    from c in customers
    join o in orders on c.ID equals o.ID
```

on 关键字在关键字段 ID 的后面, equals 关键字指定另一个集合中的对应字段。查询结果仅包含两个集中 ID 字段值相同的对象数据。

select 语句投影了一个带指定属性的新数据类型, 因此可以清楚地看到最初的总销售量、新订单和最终的新总销售量:

```
select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = o.Amount,
            SalesAfter = c.Sales+o.Amount };
```

这个程序没有在 customer 对象中递增总销售量, 但您可以在自己的业务逻辑中完成这一任务。

foreach 循环的逻辑和查询中值的显示与本章前面示例中的相同。

23.20 小结

LINQ 使 C#中编写的查询非常简单和强大。第 24 章将学习如何使用 LINQ 查询关系数据库，以高效地处理大型数据集。

方法语法中有太多的 LINQ 方法，不可能在一本入门图书中包含所有这些方法。更多的细节和示例可参阅 LINQ 的 Microsoft 联机文档。要获得每个LINQ方法的小示例，可参阅 MSDN 步骤中的 101LINQ Samples 主题(或联机查看 <http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx>)。

Eric White 在 <http://blogs.msdn.com/ericwhite/pages/FP-Tutorial.aspx> 上提供的有关函数编程的教程是学习 LINQ 中函数编程的一个好资料，该网页还提供了 LINQ 方法语法的完整教程。

23.21 练习

- (1) 修改第一个示例程序 23-1-FirstLINQquery，将结果降序排列。
- (2) 在大数程序示例 23-5-LargeNumberQuery 中修改传送给 generateLotsOfNumbers()方法的数字，创建不同规模的结果集，看看查询结果所受的影响。
- (3) 给大数程序示例 23-5-LargeNumberQuery 中的查询添加一个 orderby 子句，看看这会如何影响性能。
- (4) 修改大数程序示例 23-5-LargeNumberQuery 中的查询条件，选择数字列表中的较大和较小子集，看看这会如何影响性能？
- (5) 修改方法语法示例 23-2-LINQMethodSyntax，完全删除 where 子句，输出量会有多少？
- (6) 修改查询复杂对象程序示例 23-7-QueryComplexObjects，用对应于字段的条件选择查询字段的不同子集。
- (7) 给第一个示例程序 23-1-FirstLINQquery 添加聚合运算符，哪些简单的聚合运算符能适用于这种非数字的结果集？

附录 A 给出了练习答案。

23.22 本章要点

主 题	重 要 概 念
LINQ 的概念和使用场合	LINQ 是内置于 C#中的一种查询语言。使用 LINQ 可以在大型的对象集合、XML 或数据库中查询数据
LINQ 查询的组成部分	LINQ 查询包含 from、where、select 和 orderby 子句
获取 LINQ 查询的结果的方式	使用 foreach 语句迭代 LINQ 查询的结果
延迟执行	LINQ 查询会延迟到执行 foreach 语句时执行

(续表)

主 题	重 要 概 念
方法语法和查询语法	简单的 LINQ 查询使用查询语法，较高级的查询使用方法查询。对于任意给定的查询，查询语法和方法语法的结果相同
聚合运算符	使用 LINQ 聚合运算符获得大型数据集的信息，而无需迭代每个结果
投影	使用投影技术改变数据类型，在查询中创建新对象
组合查询	使用组合查询给数据分组，再按照组进行排序、计算聚合值以及进行比较
排序	使用 orderby 运算符给查询的结果排序
Set 运算符	使用 set 运算符 Union()、Intersect()和 Distinct()在多个结果集中查找匹配的数据
Join 运算符	使用 Join 运算符在一个查询中查找多个集合中的相关数据



# 第 24 章

## 应用 LINQ

本章内容:

---

- LINQ 变体
- 对数据库使用 LINQ
- 导航数据库关系
- 对 XML 使用 LINQ
- 使用 LINQ to XML 构造函数
- 从数据库中生成 XML
- 使用 XML 片段

第 26 章介绍了 LINQ，说明了如何使用 LINQ 处理对象。本章研究如何把 LINQ 应用于查询，处理不同数据源(例如数据库和 XML)中的数据。

### 24.1 LINQ 的变体

Visual Studio 2010 和 .NET Framework 4 带有几个内置的 LINQ 功能，它们为不同类型的数据提供了查询解决方案：

- **LINQ to Objects:** 为任意类型的 C# 内存对象提供查询，例如数组、列表和其他集合类型。上一章的所有示例都使用 LINQ to Objects。也可以把本章介绍的技巧应用于 LINQ 的所有变体。
- **LINQ to XML:** 提供了 XML 文档的创建和处理功能，其语法与一般查询机制和其他 LINQ 变体相同。
- **LINQ to ADO.NET:** .NET 的 ADO.NET 或 Active Data Objects 是一个统称，囊括了 .NET 中访问数据库(例如 Microsoft SQL Server、Oracle 等)中的数据的所有不同的类和库。LINQ to ADO.NET 包括 LINQ to Entities、LINQ to DataSet 和 LINQ to SQL。

- **LINQ to Entities:** ADO.NET Entity Framework 是 .NET 4 中数据接口类的最新集合, Microsoft 推荐使用它进行新的开发工作。本章将给 Visual C# 项目添加 ADO.NET Entity Framework 数据源, 再使用 LINQ to Entities 查询它。
- **LINQ to DataSet:** DataSet 对象在 .NET Framework 的第一版中引入。这个 LINQ 的变体允许使用 LINQ 轻松地查询旧 .NET 数据。
- **LINQ to SQL:** 这是用于 .NET 3.5 的备用 LINQ 接口, 主要用于 Microsoft SQL Server, 在 .NET 4 中被 LINQ to Entities 替代。
- **PLINQ:** PLINQ 或 Parallel LINQ 用并行编程库扩展了 LINQ to Objects, 它可以使查询同时在多核处理器上执行。

有这么多 LINQ 的变体, 根本不可能在一本入门图书中全部介绍, 所以本章介绍如何把 LINQ 应用于 XML 最常见的数据源和关系数据库实体上。LINQ 处理所有数据源的方式都是非常类似的, 所以一旦学会了如何使用两三个 LINQ 变体, 就可以很容易把 LINQ 应用于新数据源。

## 24.2 给数据库使用 LINQ

SQL 数据库(如 Microsoft SQL Server 和 Oracle)也称为关系数据库。关系数据库建立在实体-关系模型的基础上, 其中实体是数据对象(如顾客)的抽象概念, 它与其他实体(如订单和产品, 如顾客给产品下了订单)相关。

关系数据库使用 SQL 数据库语言(SQL 表示 Structured Query Language, 结构化查询语言)查询和操作数据。传统上, 处理数据库需要至少了解一些 SQL 知识, 才能把 SQL 语句嵌入编程语言, 或者把包含 SQL 语句的字符串传送给在面向 SQL 的数据库类库中的 API 调用或方法。

这听起来很复杂。现在, Visual Studio 2010 和 ADO.NET Entity Framework 可以创建 C# 对象来表示数据库模型中的实体, 再自动处理与 SQL 数据库通信的所有细节。它会把 LINQ 查询自动转换为 SQL 语句, 使程序能很容易地处理 C# 对象。

创建代码, 建立一系列匹配已有关系表结构的类和集合是很费时费力的, 但使用 LINQ to SQL 对象关系映射功能, 可以从数据库中自动创建匹配数据库表的类, 这样就不必自己创建它, 而可以直接开始使用类。

## 24.3 安装 SQL Server 和 Northwind 示例数据

要运行本章的示例, 必须安装 Microsoft SQL Server Express, 这是 Microsoft SQL Server 的轻型版本。



如果很熟悉 SQL Server, 而且能访问安装了 Northwind 示例数据库的 Microsoft SQL Server Express 2005 标准版或企业版, 就可以跳过这个安装过程, 但必须修改连接信息, 以匹配自己的 SQL Server 实例。如果没有使用过 SQL Server, 就需要安装 SQL Server Express。



### 24.3.1 安装 SQL Server Express 2008

Visual Studio 2010 和 Visual C# 2010 Express 版本都包含 SQL Server Express 副本, 这是 SQL Server 2008 的轻型桌面引擎版本。

如果已经安装了 Visual Studio 2010 或 Visual C# 2010 Express, 但没有安装 SQL Server 2008 Express Edition, 就可以使用如下 URL 下载并安装它:

<http://microsoft.com/express/sql/default.aspx>



不能使用 Microsoft SQL Server 压缩版和 LINQ to Entities, 必须使用 SQL Server 2008 Express Edition。

### 24.3.2 安装 Northwind 示例数据库

本章的示例需要 SQL Server 的 Northwind 示例数据库, 它未包含在 Visual C# 2010 或 SQL Server 2008 Express 中, 但可以从 Microsoft 获得一个独立的下载软件包。在 Google 或类似的搜索站点上搜索 “northwind sample database download”, 或者直接进入如下 URL, 就可以找到这个软件包。

[www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a68034&displaylang=en](http://www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a68034&displaylang=en)

该链接可以下载安装文件 SQL2000SampleDb.msi。

单击 Run 按钮, 执行.msi 文件, 将会安装 Northwind 示例数据库文件。接受各个安装屏幕中的默认选项。完成时, 数据库文件就安装在 C:\SQL Server 2000 Sample Databases\NORTHWND.MDF 中。



Northwind MDF 的文件名是 NORTHWND.MDF, 没有 I。

保留这个文件名和路径比较方便, 因为在以后创建与数据库的连接时要引用它。这就完成了本章需要的 SQL Express 和示例数据的安装。现在就可以使用 LINQ to Entities 了。

## 24.4 第一个 LINQ 数据库查询

在下面的示例中, 要创建一个简单的查询, 用 LINQ to SQL 找出 Northwind SQL Server 示例数据中的一个顾客对象子集, 再输出到控制台上。

试一试: 第一个 LINQ 数据库查询

按照下面的步骤在 Visual C# 2010 中创建示例:

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序项目 BegVCSharp\_24\_1\_FirstLINQtoDatabaseQuery。



(2) 单击 OK 按钮创建项目。

(3) 为了给 Northwind 数据库添加 LINQ to Entities 数据源，可进入 Solution Explorer 窗格，右击 C#项目 BegVCSharp\_24\_1\_FirstLINQtoDataQuery，选择 Data | Add New Data Source 菜单项。

(4) 在 Choose a Data Source Type 对话框中，选择 Database。

(5) 在 Choose a Database Model 对话框中，选择 Entity Data Model。

(6) 在 Choose Model Contents 对话框中，选择 Generate From Database。

(7) 在 Choose Your Data Connection 对话框中，选择 New Connection。

(8) 在 Connection Properties 对话框中，单击 Database File Name(new or existing)文本框右边的 Browse 按钮，找到安装 Northwind 数据的目录 C:\SQL Server 2000 Sample Databases\，选择 NORTHWND.MDF 作为数据库文件。单击 OK 关闭对话框，完成导入实体模型的操作。

(9) 在 Choose Your Database Objects 对话框中，展开 Tables 控件，选择 Customers、Orders 和 Order Details。单击 Finish，就可以在 labeledModel1.edmx 窗口中看到实体数据对象的图形，如图 24-1 所示。

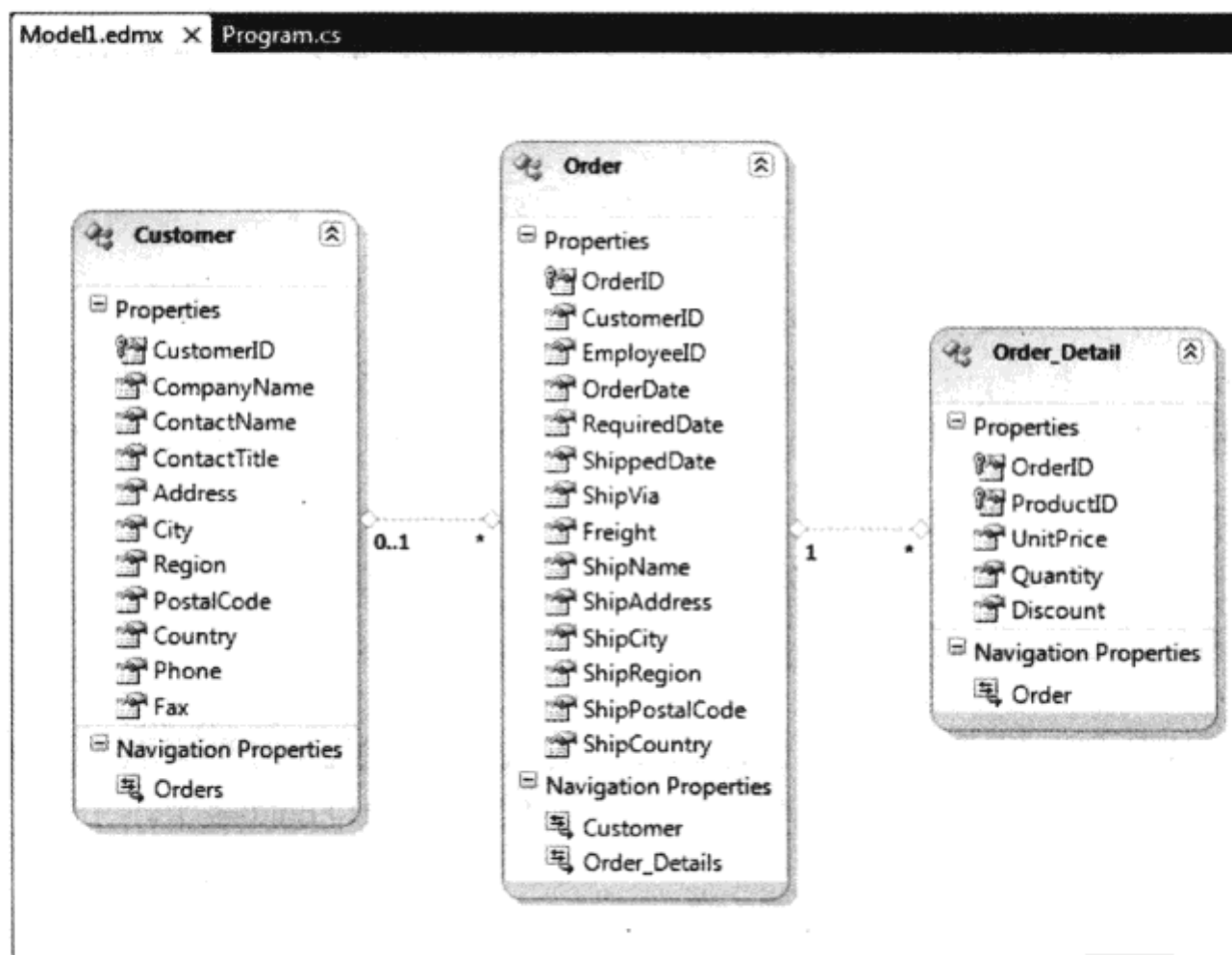


图 24-1

(10) 现在编译项目，这样在下一步开始输入代码时，就可以使用 Customer 对象了。

在 Model1.designer.cs 文件中可以看到为实体模型生成的类的代码，这些代码显示在 Solution Explorer 的 Model1.edmx 源文件的下面，类似于放在 <formname>.designer.cs 中的窗体的生成代码。与窗体的生成代码相同，不应修改设计器生成的代码，最好不要在编辑器中打开这些代码，除非要验证类的名称或选择某个生成的数据类型。

(11) 打开主源文件 Program.cs，并在 Main()方法中添加如下代码：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

    var queryResults = from c in northWindEntities.Customers
                        where c.Country == "USA"
                        select new {
                            ID=c.CustomerID,
                            Name=c.CompanyName,
                            City=c.City,
                            State=c.Region
                        };

    foreach (var item in queryResults) {
        Console.WriteLine(item);
    };

    Console.WriteLine("Press Enter/Return to continue..");
    Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp24\_1\_FirstLINQtoDatabaseQuery\Program.cs

(12) 编译并运行程序(按下 F5 键即可开始调试), 会看到来自美国的顾客信息, 如下所示:

```
{ ID = GREAL, Name = Great Lakes Food Market, City = Eugene, State = OR }
{ ID = HUNGC, Name = Hungry Coyote Import Store, City = Elgin, State = OR }
{ ID = LAZYK, Name = Lazy K Kountry Store, City = Walla Walla, State = WA }
{ ID = LETSS, Name = Let's Stop N Shop, City = San Francisco, State = CA }
{ ID = LONEP, Name = Lonesome Pine Restaurant, City = Portland, State = OR }
{ ID = OLDWO, Name = Old World Delicatessen, City = Anchorage, State = AK }
{ ID = RATTC, Name = Rattlesnake Canyon Grocery, City = Albuquerque, State = NM }
{ ID = SAVEA, Name = Save-a-lot Markets, City = Boise, State = ID }
{ ID = SPLIR, Name = Split Rail Beer & Ale, City = Lander, State = WY }
{ ID = THEBI, Name = The Big Cheese, City = Portland, State = OR }
{ ID = THECR, Name = The Cracker Box, City = Butte, State = MT }
{ ID = TRAIH, Name = Trail's Head Gourmet Provisioners, City = Kirkland,
  State = WA }
{ ID = WHITC, Name = White Clover Markets, City = Seattle, State = WA }
Press Enter/Return to continue...
```

按下回车键, 结束程序, 关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能), 就需要按下回车键两次, 这会结束程序的运行。下面详细解释一下该示例。

### 示例的说明

这个示例和本章的所有其他示例的代码都类似于第 23 章介绍 LINQ 时使用的示例。其代码都使用了 System.Linq 名称空间中的扩展类, 在创建项目时, Visual C# 2010 会自动插入一个 using 语句, 以引用这个名称空间:

```
using System.Linq;
```

使用 LINQ to Entities 类的第一步是为要访问的数据库创建ObjectContext对象的一个实例,ObjectContext是在数据源中创建的.edmx文件的编译类。这个对象是数据库的网关,提供了在程序

中控制它的所有方法。它还是创建业务对象的工厂，业务对象对应于数据库中存储的概念化实体(例如，顾客和产品)。

在项目中，数据上下文类称为 `NORTHWNDEntities`，从 `Model1.edmx` 文件中编译。`Main()` 方法的第一个操作是创建 `NORTHWNDEntities` 的一个实例，如下所示：

```
NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
```

在 **Choose Your Database Objects** 窗格中选择 **Customers** 表时，会把 **Customer** 对象添加到 `Model1.edmx` 的 **LINQ to Entities** 类中，并把 **Customers** 成员添加到 `northWindDataEntities` 对象中，以查询 **Northwind** 数据库中的 **Customer** 对象。

实际的 LINQ 查询语句把 `northWindEntities` 的 **Customers** 成员用作数据源，建立查询：

```
var queryResults = from c in northWindEntities.Customers
                    where c.Country == "USA"
                    select new {
                        ID=c.CustomerID,
                        Name=c.CompanyName,
                        City=c.City,
                        State=c.Region
                    };
```

**Customers** 是一个类型化的 LINQ 表(`System.Data.Linq.Table<Customer>`)，类似于 **Customer** 对象的类型化集合(类似于 `List<Table>`)，但它是为 LINQ to SQL 实现的，并从数据库中自动填充。它实现了 `IEnumerable/IQueryable` 接口，所以可以用作 `from` 子句中的 LINQ 数据源，这与其他集合或数组一样。

`where` 子句把结果限制为来自美国的顾客。`select` 子句是一个投影，类似于第 23 章开发的示例，它也创建了一个新对象，其成员有 **ID**、**Name**、**City** 和 **State**。由于该结果是来自美国的顾客，所以可以把 **Region** 重命名为 **State**，以便更准确地显示结果。最后创建一个标准的 `foreach` 循环，这与第 23 章编写的 `foreach` 循环类似：

```
foreach (var item in queryResults) {
    Console.WriteLine(item);
};
```

这段代码为每个 `item` 使用默认生成的 `ToString()` 方法，以便格式化 `Console.WriteLine(item)` 的输出，因此每个投影的成员示例都在花括号中显示其值：

```
{ ID=WHITC, Name=White Clover Markets, City=Seattle, State=WA }
```

最后，示例暂停显示，以便查看结果：

```
Console.WriteLine("Press Enter/Return to continue... ");
Console.ReadLine();
};
```

我们创建了一个基本的 LINQ to SQL 查询，可以将它为基础，来创建更复杂的查询。

## 24.5 浏览数据库关系

ADO.NET Entity Framework 最强大的一个方面是可以自动创建 LINQ to SQL 对象,以便浏览数据库中相关表之间的关系。下面的示例要给 LINQ to Entities 类添加一个相关表,再添加代码,以浏览数据库中的相关数据对象,并输出它们的值。

### 试一试: 浏览 LINQ to Entities 关系

按照下面的步骤在 Visual C# 2010 中创建示例:

(1) 在 C:\BegVCSharp\Chapter24 目录中修改上一个示例的项目 BegVCSharp\_24\_1\_FirstLINQto-DataQuery, 如下面的步骤所示。

(2) 打开主源文件 Program.cs。在 Main()方法中把 Orders 字段添加到 LINQ 查询的 select 子句中(别忘了在 c.Region 的后面添加一个逗号,把所添加的字段与列表的其他内容分开):

```
static void Main(string[] args)
{
    NorthwindDataContext northWindDataContext = new NorthwindDataContext();

    var queryResults = from c in northWindDataContext.Customers
                        where c.Country == "USA"
                        select new {
                            ID=c.CustomerID,
                            Name=c.CompanyName,
                            City=c.City,
                            State=c.Region,
                            Orders=c.Orders
                        };
};
```

(3) 修改 foreach 子句, 输出查询结果, 如下所示:



可从  
wrox.com  
下载源代码

```
foreach (var item in queryResults) {

    Console.WriteLine(
        "Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date",
        item.Name, item.City, item.State, item.Orders.Count
    );
    foreach (Order o in item.Orders) {
        Console.WriteLine("\t\t{0}\t{1}", o.OrderID, o.OrderDate);
    }

};

Console.WriteLine("Press Enter/Return to continue... ");
Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp\_24\_2\_NavigatingDatabaseRelationships\Program.cs

(4) 编译并执行程序(按下 F5 键即可开始调试), 会看到来自美国的顾客及其订单信息, 如下所

示(这是输出的最后一部分，第一部分已滚动出了控制台窗口的顶部)。

```
Customer: Trail's Head Gourmet Provisioners Kirkland, WA
3 orders:      Order ID      Order Date
               10574 6/19/1997 12:00:00 AM
               10577 6/23/1997 12:00:00 AM
               10822 1/8/1998  12:00:00 AM
Customer: White Clover Markets Seattle, WA
14 orders:     Order ID      Order Date
               10269 7/31/1996 12:00:00 AM
               10344 11/1/1996 12:00:00 AM
               10469 3/10/1997 12:00:00 AM
               10483 3/24/1997 12:00:00 AM
               10504 4/11/1997 12:00:00 AM
               10596 7/11/1997 12:00:00 AM
               10693 10/6/1997 12:00:00 AM
               10696 10/8/1997 12:00:00 AM
               10723 10/30/1997 12:00:00 AM
               10740 11/13/1997 12:00:00 AM
               10861 1/30/1998 12:00:00 AM
               10904 2/24/1998 12:00:00 AM
               11032 4/17/1998 12:00:00 AM
               11066 5/1/1998  12:00:00 AM
Press Enter/Return to continue...
```

与前面一样，按下回车键，结束程序，关闭控制台屏幕。

#### 示例的说明

我们修改了前面的示例，而不是从头开始创建新程序，因此不需要重复创建数据源文件 `Model1.edmx` 的步骤(注意示例代码中包含几个项目，每个项目都有自己的 `Model1.edmx` 实例)。

在 Choose Your Data Objects 对话框中选择 Orders 表，把 Order 类添加到 `Model1.edmx` 源文件中，以表示 Northwind 数据库映射中的 Orders 表。

Visual Studio 2010 检测到数据库中 Customers 和 Orders 表之间的关系，所以在 Customer 类中还添加了一个 Orders 集合成员，来表示该关系。所有这些都是自动完成的，这与给窗体添加新控件一样。

接着，在查询的 select 子句中添加新的 Orders 成员：

```
select new {
    ID=c.CustomerID,
    Name=c.CompanyName,
    City=c.City,
    State=c.Region,
    Orders=c.Orders
};
```

Orders 是一种特殊的类型化 LINQ 集(`System.Data.Linq.EntitySet<Order>`)，表示关系数据库中两个表之间的关系。它实现了 `IEnumerable/IQueryable` 接口，所以可以用作 LINQ 数据源，或者用 `foreach` 语句迭代，这与其他集合或数组一样。

与上一个示例中的 Table 对象一样，EntitySet 类似于类型化的 Order 对象集合(类似于 `List<Order>`)，但只有顾客提交的订单才会显示在该 Customer 实例的 EntitySet 成员中。

Customer 的 EntitySet 成员中的 Order 对象对应于数据库中顾客 ID 与该 Customer 实例 ID 相同的订单行。

浏览关系只需建立一个嵌套的 foreach 语句，迭代每个顾客以及每个顾客的订单即可：

```
foreach (var item in queryResults) {

    Console.WriteLine(
        "Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date",
        item.Name, item.City, item.State, item.Orders.Count
    );
    foreach (Order o in item.Orders) {
        Console.WriteLine( "\t\t{0}\t{1}", o.OrderID, o.OrderDate);
    }
};
```

这里不使用默认的 ToString() 进行格式化，而是把输出结果格式化为更容易理解的形式，这样可以在每个顾客的下面显示订单列表，正确列出其层次。格式化字符串 “Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date” 为每个顾客在第一行设置了姓名、城市和州等占位符，接着在下一行为该顾客的订单输出一个列标题。这里使用 LINQ 聚合方法 Count() 输出顾客的订单数，再在嵌套的 foreach 语句中为每一行输出订单 ID 和订单日期。

```
Customer: White Clover Markets Seattle, WA
14 orders:   Order ID Order Date
              10269 7/31/1996 12:00:00 AM
              10344 11/1/1996 12:00:00 AM
```

格式仍有待改进，因为我们希望仅显示订单的日期，而这里显示了时间。

成功查询了数据库后，下面尝试另一种数据源 XML。

## 24.6 使用 LINQ to XML

LINQ to XML 并不打算替代标准的 XML API，例如，XML DOM(Document Object Model)、XPath、Xquery 和 XSLT 等。如果熟悉这些 API 或当前需要使用或学习它们，可以继续使用或学习。

LINQ to XML 补充了这些标准 XML 类，更便于使用 XML。LINQ to XML 为创建和查询 XML 数据提供了额外的选项，代码更简单，开发许多常见的情形时速度更快，如果已经在其他程序中使用了 LINQ，开发速度将会更快。

## 24.7 LINQ to XML 函数构造方法

如前面章节所述，C# 中的一个主题是利用对象初始化和匿名类型等功能更方便地构建对象。LINQ to XML 针对这个主题引入了一种更便捷的新方式，叫作函数构建方式(functional construction)，通过这种方式可以创建 XML 文档。在这种方式中，构造函数的调用可以用反映 XML 文档结构的方式嵌套。下面的示例就使用函数构建方式建立了一个包含顾客和订单的简单 XML 文档。

## 试一试: LINQ to XML 构造函数

按照下面的步骤在 Visual Studio 2010 中创建示例:

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序 BegVCSharp\_24\_3\_LinqToXmlConstructors。

(2) 打开主源文件 Program.cs。

(3) 在 Program.cs 的开头添加对 System.Xml.Linq 名称空间的引用, 如下所示:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

(4) 在 Program.cs 的 Main()方法中添加如下代码:

```
static void Main(string[] args)
{
    XDocument xdoc = new XDocument(
        new XElement("customers",
            new XElement("customer",
                new XAttribute("ID", "A"),
                new XAttribute("City", "New York"),
                new XAttribute("Region", "North America"),
                new XElement("order",
                    new XAttribute("Item", "Widget"),
                    new XAttribute("Price", 100)
                ),
                new XElement("order",
                    new XAttribute("Item", "Tire"),
                    new XAttribute("Price", 200)
                )
            ),
            new XElement("customer",
                new XAttribute("ID", "B"),
                new XAttribute("City", "Mumbai"),
                new XAttribute("Region", "Asia"),
                new XElement("order",
                    new XAttribute("Item", "Oven"),
                    new XAttribute("Price", 501)
                )
            )
        );
    Console.WriteLine(xdoc);

    Console.Write("Program finished, press Enter/Return to continue: ");
    Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp\_24\_3\_LinqToXmlConstructors\Program.cs



(5) 编译并执行程序(按下 F5 键即可开始调试), 输出结果如下所示:

```
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

输出屏幕上显示的 XML 文档包含前面示例中顾客/订单数据的一个简化版本。注意, XML 文档的根元素是<customers>, 它包含两个嵌套的<customer>元素, 这两个元素又包含许多嵌套的<order>元素。<customer>元素有两个特性: <City>和<Region>, <order>元素也有两个特性: <Item>和<Price>。

按下回车键, 结束程序, 关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能), 就需要按回车键两次。

### 示例的说明

第一步是引用 System.Xml.Linq 名称空间。本章的所有示例都要求把这行代码添加到程序中:

```
using System.Xml.Linq;
```

在创建项目时, System.Linq 名称空间是默认包含的, 但不包含 System.Xml.Linq 名称空间, 所以必须显式地添加这行代码。

接着调用 LINQ to XML 构造函数 XDocument()、XElement()和 XAttribute(), 它们彼此嵌套, 如下所示:

```
XDocument xdoc = new XDocument(
    new XElement("customers",
        new XElement("customer",
            new XAttribute("ID", "A"),
            .
            .
            .
        )
    );
```

注意, 这些代码看起来类似于 XML 本身, 即文档包含元素, 每个元素又包含特性和其他元素。下面依次分析这些构造函数:

- XDocument(): 在 LINQ to XML 构造函数层次结构中, 最高层的对象是 XDocument(), 它表示完整的 XML 文档, 在代码中如下所示:

```
static void Main(string[] args)
{
    XDocument xdoc = new XDocument(
        .
        .
        .
    );
```

在前面的代码中, 省略了 XDocument()的参数列表, 因此可以看到 XDocument()调用在何处开头和结尾。与所有 LINQ to XML 构造函数一样, XDocument()也把对象数组(object[])作为它的参数, 以便给它传送其他构造函数创建的其他对象。在这个程序中调用的所有其他构造函数都是

XDocument()构造函数的参数。这个程序传送的第一个也是唯一一个参数是 XElement()构造函数。

- XElement(): XML 文档必须有一个根元素, 所以大多数情况下, XDocument()的参数列表都以一个 XElement 对象开头。XElement()构造函数把元素名作为字符串, 其后是包含在该元素中的一个XML 对象列表。本例中的根元素是 customers, 它又包含一个 customer 元素列表:

```
new XElement("customers",
    new XElement("customer",
        .
        ),
    .
    )
```

customer 元素不包含其他 XML 元素, 只包含 3 个 XML 特性, 它们用 XAttribute()构造函数构建。

- XAttribute(): 这里给 customer 元素添加了 3 个 XML 特性: ID、City 和 Region:

```
new XAttribute("ID", "A"),
new XAttribute("City", "New York"),
new XAttribute("Region", "North America"),
```

根据定义, XML 特性是一个 XML 叶节点, 它不包含其他 XML 节点, 所以 XAttribute()构造函数的参数只有特性的名称和值。本例中生成的 3 个特性是 ID="A"、City="New York"和 Region="North America"。

- 其他 LINQ to XML 构造函数: 这个程序中没有调用它们, 但所有的 XML 节点类型都有其他 LINQ to XML 构造函数, 例如, XDeclaration()用于 XML 文档开头的 XML 声明, XComment()用于 XML 注释等。这些构造函数不太常用, 但如果需要它们来精确控制 XML 文档的格式化, 就可以使用它们。

下面继续解释第一个示例: 在 customer 元素的 ID、City 和 Region 特性后面再添加两个子 order 元素:

```
new XElement("order",
    new XAttribute("Item", "Widget"),
    new XAttribute("Price", 100)
),
new XElement("order",
    new XAttribute("Item", "Tire"),
    new XAttribute("Price", 200)
)
```

这些 order 元素都有两个特性: Item 和 Price, 但没有子元素。

接着将 XDocument 的内容显示在控制台屏幕上:

```
Console.WriteLine(xdoc);
```

这行代码使用 XDocument()的默认方法 ToString()输出 XML 文档的文本。

最后暂停屏幕, 以查看控制台输出, 再等待用户按下回车键:

```
Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
```

程序退出 Main()方法后, 就结束程序。

### 用字符串构建 XML 元素的文本

刚才执行的示例格式化了 XML，其元素中没有文本内容。XML 常常包含文本内容，使用 LINQ to XML 构造函数 `XElement()` 很容易格式化 XML 元素。例如，为了不把 `<customer>` 元素的 ID 文本建立为特性，只需把一个字符串作为构造函数 `XElement()` 的参数传入，而不是传入嵌套的 `XAttribute`：

```
XDocument xdoc = new XDocument(
    new XElement("customers",
        new XElement("customer",
            "AAAAAA",
            new XAttribute("City", "New York"),
            new XAttribute("Region", "North America")
        ),
        new XElement("customer",
            "BBBBBB",
            new XAttribute("City", "Mumbai"),
            new XAttribute("Region", "Asia")
        )
    );
```

这会生成如下所示的 XML 文档：

```
<customers>
  <customer City="New York"Region="North America"> AAAAAA </customer>
  <customer City="Mumbai"Region="Asia"> BBBBBB </customer>
</customers>
```

构造函数 `XElement()` 把参数列表中的所有字符串联起来，放在元素的文本部分。

## 24.8 保存和加载 XML 文档

注意，用 `Console.WriteLine()` 把 XML 文档显示到控制台屏幕上时，不会显示以 `<?xml version="1.0"` 开头的 XML 声明。可以用 `XDeclaration()` 构造函数显式地创建这样一个声明，但一般不需要这么做，因为在用 LINQ to XML 方法 `Save()` 把 XML 文档保存到文件中时，会自动创建该声明。

另外，在程序中构建 XML 文档有助于理解构造函数的工作方式，但我们不经常这么做，而是从外部的数据源(如文件)中加载 XML 文档。

下面的示例将试验这两个操作。

### 试一试：保存和加载 XML 文档

按照下面的步骤在 Visual Studio 2010 中创建示例：

- (1) 在 `C:\BegVCSharp\Chapter24` 目录中修改上一个示例或者创建一个新的控制台应用程序 `BegVCSharp_24_4_SaveLoadXML`。
- (2) 打开主源文件 `Program.cs`。
- (3) 在 `Program.cs` 的开头添加对 `System.Xml.Linq` 名称空间的引用，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Xml.Linq;
using System.Text;
```

如果正在修改上一个示例，则已经引用了这个名称空间。

(4) 把上一个示例中的 XML 文档构造函数及其嵌套的 XML 元素和特性调用添加到 Program.cs 的 Main()方法中：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    XDocument xdoc = new XDocument(
        new XElement("customers",
            new XElement("customer",
                new XAttribute("ID", "A"),
                new XAttribute("City", "New York"),
                new XAttribute("Region", "North America"),
                new XElement("order",
                    new XAttribute("Item", "Widget"),
                    new XAttribute("Price", 100)
                ),
                new XElement("order",
                    new XAttribute("Item", "Tire"),
                    new XAttribute("Price", 200)
                )
            ),
        new XElement("customer",
            new XAttribute("ID", "B"),
            new XAttribute("City", "Mumbai"),
            new XAttribute("Region", "Asia"),
            new XElement("order",
                new XAttribute("Item", "Oven"),
                new XAttribute("Price", 501)
            )
        )
    );
};
```

代码段 BegVCSharp\Chapter24\BegVCSharp\_24\_4\_SaveLoadXml\Program.cs

(5) 在上一步中添加了 XML 文档构造函数代码后，在 Program.cs 的 Main()方法最后添加下面的代码，以便保存、加载和显示 XML 文档：

```
string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example2.xml";

xdoc.Save(xmlFileName);

XDocument xdoc2 = XDocument.Load(xmlFileName);

Console.WriteLine("Contents of xdoc2: ");
Console.WriteLine(xdoc2);

Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
}
```

(6) 编译并执行程序(按下 F5 键即可开始调试), 控制台窗口中的输出结果如下所示:

```
Contents of xdoc2:
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget"Price="100"/>
    <order Item="Tire"Price="200"/>
  </customer>
  <customer ID="B"City="Mumbai"Region="Asia">
    <order Item="Oven"Price="501"/>
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

按下回车键, 结束程序, 关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能), 就需要按回车键两次。

#### 示例的说明

与前面一样, 第(1)步也是引用 System.Xml.Linq 名称空间。接着是对 LINQ to XML 构造函数 XDocument()、XElement()和 XAttribute()的嵌套调用。请参阅第一个示例中对这部分和其他重复代码的说明。

创建了 XDocument()对象后, 把文件名指定为一个字符串, 调用 Save()方法把 XML 文档保存到文件中:

```
string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example2.xml";

xdoc.Save(xmlFileName);
```

在这个例子中, XML 文档保存到指定的文件中, Save()方法还有一些重载版本, 可以保存到 System.IO.TextWriter 或 System.Xml.XmlWriter 中, 如果要编写另一个程序, 且该程序已经使用其中的一个类写入到文件, 就可以使用这些重载版本。

Save()方法的一个重载版本还可以指定 SaveOptions, 以便禁用格式化功能(XML 文档保存时, 默认带有缩进和空白, 使文档看起来很“美观”)。

把文档保存到文件中后, 就把它加载到一个新的 XDocument 实例 xdoc2 中:

```
XDocument xdoc2 = XDocument.Load(xmlFileName);
```

XDocument.Load()方法是静态的, 因为它是一个工厂类型的方法, 可以创建 XDocument 的新实例; 使用它可以加载由另一个完全不同的程序创建的文档。

之后显示文档, 这与前面一样, 只是这次使用从文件中加载的 xdoc2 实例。程序的剩余部分与上一个示例中的相同:

```
Console.WriteLine("Contents of xdoc2: ");
Console.WriteLine(xdoc2);

Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
```

### 24.8.1 从字符串中加载 XML

有时并不是从文件中加载XML，而是通过一个或多个方法接收从另一个应用程序发送的字符串形式的XML。在LINQ to XML中，可以使用Parse()方法通过字符串创建XML文档：

```
XDocument xdoc = XDocument.Parse(@"
  <customers>
    <customer ID= \"A\" \" City= \"New York\" \" Region= \"North America\" \">
      <order Item= \"Widget\" \" Price= \"100\" \" />
      <order Item= \"Tire\" \" Price= \"200\" \" />
    </customer>
    <customer ID= \"B\" \" City= \"Mumbai\" \"Region= \"Asia\" \">
      <order Item= \"Oven\" \" Price= \"501\" \" />
    </customer>
  </customers>
");
```

这段代码生成的结果与从文件中加载文档一样。Parse()方法与Load()方法一样，也是一个类级方法，可以创建XDocument的新实例。在调用Parse()方法之前，不需要创建新的XDocument对象。



在前面的示例中，XML的字符串字面量带双引号，但在字符串的实际内容中，并没有使用双引号。双引号只能用于在@引用的字符串字面量中包含引号。

### 24.8.2 已保存的XML文档内容

用IE打开刚才在上一个示例中保存的文档。在地址栏中指定完整的路径名C:\BegVCSharp\Chapter24\Xml\example2.xml。

注意，XML文档声明<?xml version="1.0" encoding="utf-8"?>位于所保存文档的开头，但用Console.WriteLine()把XDocument对象显示在屏幕上时，该声明不会显示。使用LINQ to XML提供的默认功能时，不需要担心声明和许多其他XML细节。



在Windows中，XML文档的默认编码方式是UTF-8(8位Unicode Transformation Format)。不应改变这种编码方式，除非在特别不寻常的情况下，例如，创建一个ASCII编码的XML文档，由不理解UTF-8的旧程序使用。此时，可以添加一个XDeclaration()对象，在XDocument()构造函数的参数列表开头，把编码方式设置为ASCII，或者设置XDocument的Declaration属性：

```
xdoc.Declaration = new XDeclaration("1.0", "us-ascii", "yes");
```

## 24.9 处理XML片段

与一些XML API不同，LINQ to XML处理XML片段(部分或不完整的XML文档)的方式与处理完整的XML文档几乎完全相同。在处理片段时，只是把XElement(而不是XDocument)当作顶级



XML 对象。



唯一的限制是不能添加某些比较深奥的 XML 节点类型，XML 文档只能把这些节点类型应用于 XML 文档或 XML 片断，例如，XComment 应用于 XML 注释，XDeclaration 应用于 XML 文档声明，XProcessingInstruction 用于 XML 处理指令。

下面的示例会加载、保存和处理 XML 元素及其子节点，这与处理 XML 文档一样。

#### 试一试：处理 XML 片段

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中修改上一个示例或者创建一个新的控制台应用程序 BegVCSharp\_24\_5\_XMLFragments。

(2) 打开主源文件 Program.cs。

(3) 在 Program.cs 开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

如果正在修改上一个示例，则已经引用了这个名称空间。

(4) 把上一个示例中的 XML 元素(不包含 XML 文档构造函数)添加到 Program.cs 的 Main()方法中：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    XElement xcust =
        new XElement("customers",
            new XElement("customer",
                new XAttribute("ID", "A"),
                new XAttribute("City", "New York"),
                new XAttribute("Region", "North America"),
                new XElement("order",
                    new XAttribute("Item", "Widget"),
                    new XAttribute("Price", 100)
                ),
            new XElement("order",
                new XAttribute("Item", "Tire"),
                new XAttribute("Price", 200)
            ),
            new XElement("customer",
                new XAttribute("ID", "B"),
                new XAttribute("City", "Mumbai"),
                new XAttribute("Region", "Asia"),
                new XElement("order",
                    new XAttribute("Item", "Oven"),
```



```

        new XAttribute("Price", 501)
    )
}
;

```

---

代码段 BegVCSharp\Chapter24\BegVCSharp\_24\_5\_XmlFragments\Program.cs

---

(5) 在上一步添加了XML元素构造函数代码后,添加下面的代码,以便保存、加载和显示XML元素:

```

string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example3.xml";
xcust.Save(xmlFileName);

XElement xcust2 = XElement.Load(xmlFileName);

Console.WriteLine("Contents of xcust: ");
Console.WriteLine(xcust);

Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
}

```

(6) 编译并执行程序(按下 F5 键即可开始调试),控制台窗口中的输出结果如下所示:

```

Contents of XElement xcust2:
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:

```

按下回车键,结束程序,关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能),就需要按下回车键两次。

#### 示例的说明

XElement 和 Xdocument 都继承自 LINQ to XML 类 XContainer, 它实现了一个可以包含其他 XML 节点的 XML 节点。这两个类都实现了 Load() 和 Save() 方法, 因此, 可以在 LINQ to XML 的 XDocument 上执行的大多数操作都可以在 XElement 实例及其子元素上执行。

这里只创建了一个 XElement 实例, 它的结构与前面示例中的 XDocument 相同, 但不包含 XDocument。这个程序的所有操作处理的都是 XElement 片段。

XElement 还支持 Load() 和 Parse() 方法, 可以分别从文件和字符串中加载 XML。

---

## 24.10 从数据库中生成 XML

XML 常常用于在客户机和服务器之间交流数据，或者在多层应用程序的不同层之间交流数据。在数据库中查询某些数据，再根据这些数据生成 XML 文档或片段，传送给另一层是很常见的。下面的示例就创建一个查询，在 Northwind 示例数据库中查找某些数据，用 LINQ to SQL 查询数据，再用 LINQ to XML 类把数据转换为 XML。

### 试一试：从数据库生成 XML

按照下面的步骤在 Visual Studio 2010 中创建示例：

(1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序 BegVCSharp\_24\_6\_XML-fromDatabase。

(2) 按照本章开头的“第一个 LINQ 数据库查询”示例，在项目中添加一个新数据源 Model1.edmx，再添加与 Northwind 示例数据库的连接。

(3) 编译项目，这样在 Model1.edmx 中定义的和属性就可以在后面编辑代码时通过 IntelliSense 使用了。

(4) 打开主源文件 Program.cs。

(5) 在 Program.cs 开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

(6) 把下面的代码添加到 Program.cs 的 Main()方法中：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

    XElement northwindCustomerOrders =
        new XElement("customers",
            from c in northWindEntities.Customers.AsEnumerable()
            select new XElement("customer",
                new XAttribute("ID", c.CustomerID),
                new XAttribute("City", c.City),
                new XAttribute("Company", c.CompanyName),
                from o in c.Orders
                select new XElement("order",
                    new XAttribute("orderID", o.OrderID),
                    new XAttribute("orderDay",
                        o.OrderDate.Value.Day),
                    new XAttribute("orderMonth",
                        o.OrderDate.Value.Month),
                    new XAttribute("orderYear",
                        o.OrderDate.Value.Year),
                    new XAttribute("orderTotal",
                        o.Order_Details.Sum(od => od.Quantity * od.UnitPrice))
                ) //end order
            )
```

```

        ) // end customer
    ); // end customers'

    string xmlFileName =
        @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
    northwindCustomerOrders.Save(xmlFileName);

    Console.WriteLine("Successfully saved Northwind customer orders to:");
    Console.WriteLine(xmlFileName);
    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}

```

---

代码段 BegVCSharp\Chapter24\BegVCSharp\_24\_6\_XMLfromDatabase\Program.cs

---

#### (7) 编译并执行程序(按下 F5 键即可开始调试), 结果如下所示:

```

Successfully saved Northwind customer orders to:
C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml
Program finished, press Enter/Return to continue:

```

按下回车键, 结束程序, 关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能), 就需要按下回车键两次。

#### 示例的说明

在 Program.cs 中添加了对 System.Xml.Linq 名称空间的引用, 所以可以调用 LINQ to XML 构造函数类。

如本章第一部分所述, 我们为 Northwind 示例数据库创建了一个数据源, 再使用 VS2010 为 Northwind 数据创建一个 LINQ to Entities 对象模型。在主程序中, 创建了一个 Northwind 数据上下文类的实例, 以使用如下映射:

```
NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
```

LINQ to Entities 查询把 Northwind 数据上下文对象的 Customers 成员作为数据源, 通过 Customers、Orders 和 Order Details 表生成包含了所有顾客订单的列表。但是, 由于 LINQ to Entities 查询的延迟执行, 我们使用 Customer 对象上的 AsEnumerable() 方法把中间结果转换为内存中的 LINQ to Object 可枚举类型。最后, 查询结果被投影到查询的 Select 子句中, 成为一组嵌套的 LINQ to XML 元素和特性:

```

XElement northwindCustomerOrders =
    new XElement("customers",
        from c in northWindDataContext.Customers.AsEnumerable()
        select new XElement("customer",
            new XAttribute("ID", c.CustomerID),
            new XAttribute("City", c.City),
            new XAttribute("Company", c.CompanyName),
            from o in c.Orders
            select new XElement("order",
                new XAttribute("orderID", o.OrderID),
                new XAttribute("orderDay",
                    o.OrderDate.Value.Day),

```

```

        new XAttribute("orderMonth",
            o.OrderDate.Value.Month),
        new XAttribute("orderYear",
            o.OrderDate.Value.Year),
        new XAttribute("orderTotal",
            o.Order_Details.Sum(od => od.Quantity * od.UnitPrice))
    ) //end order
) // end customer
); // end customers

```

为了提取顾客的所有订单，在第一个 LINQ 查询(from c in northwindDataContext. Customers...) 中嵌套使用了第二个查询(from o in c.Orders...).

把 OrderDate 字段分为月份、日期和年份部分，可以使 XML 更容易查询。下一个示例将说明如何完成这项任务。

最后，把生成的 XML 保存到文件中，这与上一个示例相同：

```

string xmlFileName =
    @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
northwindCustomerOrders.Save(xmlFileName);

Console.WriteLine("Successfully saved Northwind customer orders to:");
Console.WriteLine(xmlFileName);

```

下面准备针对刚才写入磁盘的 XML 文件编写一个查询。

## 24.11 查询 XML 文档的方法

为什么需要在 XML 文档上执行 LINQ 查询？如果程序接收由另一个程序生成的 XML，就需要在该 XML 中查找特定的 XML 元素或特性，来确定如何处理它。程序可能只关注 XML 元素的一个子集，也可能需要对文档中的元素计数，或者搜索满足某个条件的元素或特性。LINQ 查询是针对这类情况的强大解决方案。

为了查询 XML 文档，LINQ to XML 类(如 XDocument 和 XElement)提供了成员属性和方法，它们返回 XML 文档或片段中可以使用 LINQ 查询的、由该 LINQ to XML 类表示的 LINQ to XML 对象集合。

下面的示例在前面示例创建的文档上使用了这些可查询的成员方法和属性。

### 试一试：查询 XML 文档

按照下面的步骤在 Visual Studio 2010 中创建示例：

- (1) 在 C:\BegVCSharp\Chapter24 目录中创建一个新的控制台应用程序 BegVCSharp\_24\_7\_QueryXML。
- (2) 打开主源文件 Program.cs。
- (3) 在 Program.cs 的开头添加对 System.Xml.Linq 名称空间的引用，如下所示：

```

using System;
using System.Collections.Generic;

```

```
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

(4) 把下面的代码添加到 Program.cs 的 Main()方法中:



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    string xmlFileName= @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
    XDocument customers = XDocument.Load(xmlFileName);

    Console.WriteLine("Elements in loaded document: ");
    var queryResult = from c in customers.Elements()
                      select c.Name;
    foreach (var item in queryResult)
    {
        Console.WriteLine(item);
    }
    Console.Write("Press Enter/Return to continue: ");
    Console.ReadLine();
}
```

代码段 BegVCSharp\Chapter24\BegVCSharp\_24\_7\_QueryXML\Program.cs

(5) 编译并执行程序(按下 F5 键即可开始调试), 结果如下所示:

```
Elements in loaded document:
customers
Press Enter/Return to continue:
```

(6) 按下回车键, 结束程序, 关闭控制台屏幕。如果使用Ctrl+F5 组合键(启动时不使用调试功能), 就需要按下回车键两次。

#### 示例的说明

在读取每个查询方法的说明时, 修改了刚才创建的 LINQ to XML 查询示例, 以使用它。每个查询都返回包含 Name 属性的LINQ to XML 元素或特性对象集合, 所以 select 子句仅返回这个 Name 属性, 并在 foreach 循环中输出它:

```
var queryResult = from c in customers.Elements()
                  select c.Name;
foreach (var item in queryResult)
{
    Console.WriteLine(item);
}
```

在开发或调试程序时, 可以首先使用此类代码来查看查询返回了什么结果。后面将修改输出, 以便显示对终端用户更有意义的业务结果。

## 24.12 使用 LINQ to XML 查询成员

本节介绍可用的LINQ to XML 查询成员, 然后把 NorthwindCustomerOrders.xml 文件作为数据源,

依次试验它们。

### 24.12.1 Elements()

第一个 LINQ to XML 查询方法是 XDocument 类的 Elements() 成员, 它也可以用于 XElement 类。

Elements() 返回 XML 文档或片段中的所有第一级元素。对于有效的 XML 文档, 例如, 刚才创建的 NorthwindCustomerOrders.xml 文件, 只有一个第一级元素, 即根元素 customers:

```
< ?xml version="1.0" encoding="utf-8" ? >
< customers >
.
< /customers >
```

其他元素都是 customers 的子元素, 所以 Elements() 只返回一个元素:

```
Elements in loaded document:
customers
```

XML 片段可能包含多个第一级元素, 但查询子元素通常更有用, 这需要使用下一节介绍的 Descendants() 成员。

### 24.12.2 Descendants()

下一个 LINQ to XML 查询方法是 XDocument 类的 Descendants() 成员, 它也可以用于 XElement 类。

Descendants() 返回 XML 文档或片段中的所有子元素(所有级别的子元素)。修改示例 BegVCSharp\_24\_7-QueryXML, 如下所示:

```
Console.WriteLine("All descendants in document: ");
queryResult =
    from c in customers.Descendants()
    select c.Name;
foreach (var item in queryResult)
{
    Console.WriteLine(item);
}
```

编译并执行上述代码, 会发现 customer 和 order 元素名以文档中的顺序重复出现:

```
All descendants in document:
customer
order
order
.
customer
order
.
customer
order
.
order
order
Press Enter/Return to continue:
```

这个输出滚动出了屏幕，所以可能看不到第一部分。这说明，NorthwindCustomerOrders.xml 文件只在根元素 customers 下包含 customer 和 order 元素：

```
<?xml version="1.0" encoding="utf-8" ?>
<customers >
  <customer . [[[SPACE]]]>
    <order . />
    <order . />
  .
</customer>
<customer ...>
  <order . />
.
```

在结果上添加 LINQ 运算符 Distinct()，这样会更便于管理输出：

```
Console.WriteLine("All distinct descendants in document: ");
var queryResult =
    from c in customers.Descendants()
    select c.Name;
foreach (var item in queryResult.Distinct())
```

结果，列表中只包含不同的元素名：

```
All distinct descendants in document:
customers
customer
order
Press Enter/Return to continue:
```

第一次处理文档时，这非常有助于研究文档的结构。但查找所有元素并不是常常需要在已完成的应用程序中解决的问题。

一种常见的情形是查找具有指定名称的子级元素。Descendants()方法的一个重载版本把需要的元素名作为字符串参数，如下所示：

```
Console.WriteLine("Descendants named 'customer': ");
var queryResult =
    from c in customers.Descendants("customer")
    select c.Name;
foreach (var item in queryResult) // remove Distinct()
{
    Console.WriteLine(item);
}
```

这会返回 customer 元素：

```
Descendants named 'customer':
customer
customer
customer
.
customer
customer
Press Enter/Return to continue:
```

显然，这是一个更加通用的查询。查询指定类型的一组元素后，就可以搜索特定的特性，如下



所示。



为了完整起见, LINQ to XML 还提供了一个 `Ancestors()` 方法, 它与 `Descendants()` 方法相反, 在 XML 文档的树形结构中返回比源元素级别高的所有元素。它不如 `Descendants()` 方法常用, 因为开发人员常常从根元素开始处理 XML 文档, 沿着元素和属性的树形结构向下查找, 一直到叶节点一级。Parent 特性指向直接上级节点, 比较常用。

### 24.12.3 Attributes()

下一个 LINQ to XML 查询方法是 `Attributes()` 成员, 它返回当前选中元素的所有特性。为了说明它的工作方式, 修改 `BegVCSharp_24_7_QueryXML` 示例, 如下所示:

```
Console.WriteLine("Attributes of descendants named 'customer': ");
var queryResult =
    from c in customers.Descendants("customer").Attributes()
    select c.Name;
foreach (var item in queryResult)
{
    Console.WriteLine(item);
}
```

编译并执行代码, 会看到 `customer` 元素的特性名:

```
Attributes of descendants named 'customer':
ID
City
Company
ID
City
Company
.
ID
City
Company
ID
City
Company
Press Enter/Return to continue:
```

这个输出再次滚动出了屏幕。这个查询找出了 `customer` 元素的特性名:

```
< customer ID= . City= . Company= . >
< customer ID= . City= . Company= . >
< customer ID= . City= . Company= . >
.
```

与 `Descendants()` 方法一样, 也可以给 `Attributes()` 传送要搜索的具体名称。另外, 不仅可以显示名称, 还可以显示特性本身。下面的查询显示顾客 `Company` 的特性:

```
Console.WriteLine("customer attributes named 'Company': ");
```

```

var queryResult =
    from c in customers.Descendants("customer").Attributes("Company")
    select c;
foreach (var item in queryResult)
{
    Console.WriteLine(item);
}

```

编译并运行代码，就会看到 customer 元素的 Company 特性：

```

.
Company="Toms Spezialitäten"
Company="Tortuga Restaurante"
Company="Tradição Hipermercados"
Company="Trail's Head Gourmet Provisioners"
Company="Vaffeljernet"
Company="Victuailles en stock"
Company="Vins et alcools Chevalier"
Company="Die Wandernde Kuh"
Company="Wartian Herkku"
Company="Wellington Importadora"
Company="White Clover Markets"
Company="Wilman Kala"
Company="Wolski Zajazd"
Press Enter/Return to continue:

```

下面是另一个示例，这次显示 orders 元素和 orderYear 特性：

```

Console.WriteLine("order attributes named 'orderYear': ");
var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")
    select c;
foreach (var item in queryResult)
{
    Console.WriteLine(item);
}

```

编译并执行，结果如下：

```

.
orderYear="1998"
orderYear="1998"
orderYear="1998"
orderYear="1996"
orderYear="1997"
orderYear="1997"
orderYear="1998"
orderYear="1998"
orderYear="1998"
orderYear="1998"
Press Enter/Return to continue:

```

还可以用 Value 属性获得特性的值(这里显示年份)：

```

Console.WriteLine("Values of order attributes named 'orderYear': ");
var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")

```

```

        select c.Value;
    foreach (var item in queryResult)
    {
        Console.WriteLine(item);
    }

```

编译并运行，结果如下：

```

.
1996
1997
1997
1998
1998
1998
1998
Press Enter/Return to continue:

```

现在可以回答一个问题：订单最早是在哪一年下的？这可以使用相同的查询，但需要在结果上应用 `Min()` 聚合运算符，而不使用常用的 `foreach` 循环：

```

var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")
    select c.Value;
Console.WriteLine("Earliest year in which orders were placed: {0}",
    queryResults.Min());

```

编译并执行，结果是 1996 年：

```

Earliest year in which orders were placed: 1996
Press Enter/Return to continue:

```

本章的练习要求回答指定的问题。

## 24.13 小结

至此，就完成了对 LINQ to XML 的介绍。LINQ to XML 集成了 LINQ 的概念和易用的 XML API，把 XML 快速集成到其他使用 LINQ 的其他程序中。因此，对于熟悉 LINQ 的其他形式的程序员而言，XML 文档上的查询非常简单、自然。

本章学习了如何用 LINQ to XML 函数构建方式构建 XML 文档，如何用 LINQ to XML 加载和保存 XML 文档。

我们掌握了用 LINQ to XML 处理不完整的 XML 文档(片段)，学习了如何从 LINQ to SQL 或 LINQ to Objects 查询中生成 XML 文档。

最后探讨了如何用 LINQ to XML 查询已有的 XML 文档，如何在 LINQ to XML 中使用高级 LINQ 特性，如 LINQ 聚合运算符。

## 24.14 练习

(1) 用 LINQ to XML 构造函数创建下面的 XML 文档：

```
<employees>
  <employee ID="1001" FirstName="Fred" LastName="Lancelot">
    <Skills>
      <Language> C# </Language>
      <Math> Calculus </Math>
    </Skills>
  </employee>
  <employee ID="2002" FirstName="Jerry" LastName="Garcia">
    <Skills>
      <Language> French </Language>
      <Math> Business </Math>
    </Skills>
  </employee>
</employees>
```

- (2) 为自己创建的 NorthwindCustomerOrders.xml 文件编写一个查询，查找最早购买产品的顾客 (在 Northwind 运行的第一年即 1996 年下订单的顾客)。
  - (3) 为 NorthwindCustomerOrders.xml 文件编写一个查询，查找单份订单金额超过 10 000 美元的顾客。
  - (4) 为 NorthwindCustomerOrders.xml 文件编写一个查询，查找购买量最大的顾客，例如，订单总额超过 100 000 美元的公司。
  - (5) 使用 LINQ to Entities 显示 Northwind 数据库中 Products 和 Employees 表的详细信息。
  - (6) 创建一个 LINQ to Entities 查询，显示 Northwind 数据库中最畅销的产品。
  - (7) 创建一个组合查询，按国家显示最畅销的产品。
- 附录 A 给出了练习答案。

24.15 本章要点

主 题	重 要 概 念
不同的 LINQ 变体	.NET 中每个不同的数据源都有一个 LINQ 变体或“风格”，可用于查询其数据
如何使用 LINQ 查询数据库	使用 Visual C# 2010 中的 Data Source Configuration Wizard(选择 Data   Add New Data Source)，可以为数据库生成 LINQ to Entities 类
如何使用 LINQ 导航数据库关系	LINQ to Entities 类为每个添加到数据源中的相关数据实体(表)包含了可导航的实例成员
如何使用 LINQ 轻松地构建 XML	LINQ to XML 包含非常强大的函数构造方式，用于从任何 LINQ 查询中生成 XML 文档
如何从数据库中创建 XML	在一个查询中组合 LINQ to Entities、LINQ to Objects 和 LINQ to XML，可以从数据库中构建 XML
如何创建 XML 文件和片段	使用 LINQ to XML 包含的方法，可以轻松地加载文件中的 XML，把 XML 保存到文件中，以及操作 XML 文档的各个部分

# 第 V 部分

## 其 他 技 术

---

- 第 25 章 Windows Presentation Foundation
- 第 26 章 Windows Communication Foundation
- 第 27 章 Windows Workflow Foundation



# 第25章

## Windows Presentation Foundation

### 本章内容:

---

- WPF 的含义
- 基本 WPF 应用程序的组成
- WPF 基础
- 用 WPF 编程

本书介绍了应用程序的两种主要类型：用户直接运行的桌面应用程序，以及用户通过浏览器访问的 Web 应用程序。我们在 .NET Framework 的两个不同区域创建它们：Windows 窗体和 ASP.NET 页面。这些应用程序类型有各自的优缺点。桌面应用程序有较大的灵活性和较强的响应能力，而 Web 应用程序可以被许多用户同时在远程访问。

在目前的计算环境下，应用程序之间的界限越来越模糊了。有了 Web 服务和 Windows Communication Foundation(WCF，详见第 26 章)后，桌面应用程序和 Web 应用程序都可以用更分布的方式执行，在局域网和广域网上交换数据。另外，Web 客户应用程序(即 IE 或 Firefox 等浏览器)不再被看作所谓的“瘦”客户程序，因为它们不再仅仅显示信息。最新的浏览器和运行它们的计算机的功能已经远不止此。

近年来，用户体验大有渐趋融合之势。Web 应用程序现在一般使用 JavaScript、Flash、Java 应用程序和其他技术，越来越像桌面应用程序了。例如，Google Docs 的功能就说明了这一点。而桌面应用程序相互连接的趋势越来越明显，其功能从简单(自动更新、联机帮助等)到高级(例如，联机数据源和对等联网)，应有尽有。这一点可以通过图 25-1 加以说明。

Windows Presentation Foundation(WPF)是一种统一的技术，利用它编写的应用程序可以在桌面和 Internet 之间搭起桥梁。本章介绍的 WPF 应用程序可以作为桌面应用程序运行，或在浏览器上作为 Web 应用程序运行。WPF 有一个删节版本 Silverlight，可用于给 Web 应用程序添加动态内容。

本章将学习 WPF，理解如何使用它创建下一代应用程序。



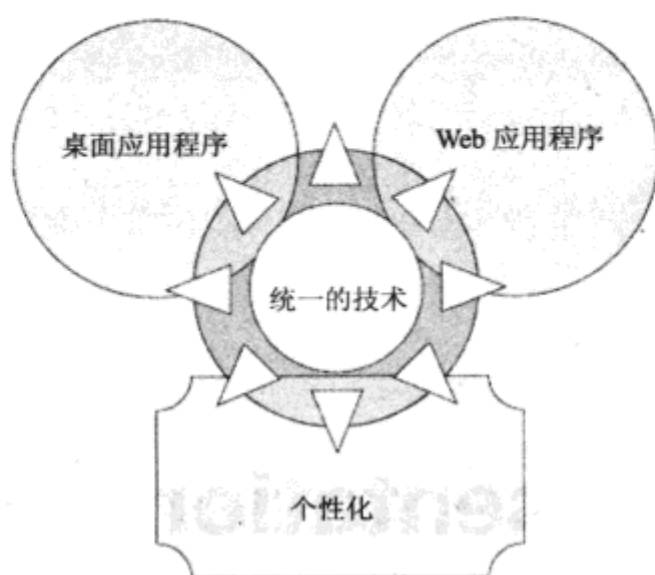


图 25-1

## 25.1 WPF 的概念

利用 WPF(以前称为 Avalon)技术可以编写出独立于平台的应用程序,它清晰地定义了设计和功能之间的界限。WPF 借用并扩展了以前许多技术的概念和类,包括 Windows 窗体、ASP.NET、XML、数据绑定技术和 GDI+等。如果读者以前使用 .NET Framework 创建过 Web 应用程序,第一次看到 WPF 应用程序的代码时,会立即注意到许多类似之处。尤其是, WPF 应用程序使用标记和代码隐藏模型,这与 ASP.NET 类似。但在其内部,它们的不同点和相似点一样多,因此 WPF 对于开发人员 and 用户而言都是一种全新的技术。

WPF 开发的一个关键概念是设计与功能几乎完全分开。这样,设计人员和 C#开发人员就可以一起开发项目,而在之前,要达到这样的灵活程度,需要使用高级设计概念或第三方工具。这个功能受到所有人的欢迎,包括小型团队、爱好开发的人员、合作开发大项目的大型开发团队和设计人员。

下面几节介绍 WPF 给设计人员和开发人员带来了什么好处,使他们能携手工作。

### 25.1.1 WPF 给设计人员带来的好处

在 WPF 中,用户界面设计使用的语言是 Extensible Application Markup Language(XAML,读作 zammel)。它类似于 ASP.NET 中使用的标记语言,因为它也使用 XML 语法,允许以声明性的、带层次结构的方式把控件添加到用户界面上。也就是说,可以用 XML 元素的形式添加控件,用 XML 特性指定控件的属性。控件也可以包含其他控件,这对布局和功能都非常重要。

但是, XAML 比 ASP.NET 强大许多,在显示给用户时绝不仅限于 HTML 的功能。XAML 在设计时考虑了目前强大的图形卡,并允许使用这些图形卡通过 DirectX 7 或更高版本提供的全部高级功能。下面列出了这些功能:

- 浮点数坐标和矢量图提供的布局可以缩放、旋转和变换,且没有质量损失。
- 2D 和 3D 高级渲染功能
- 字体的高级处理和渲染

- UI 对象的纯色、渐变色和纹理填充，且可以设置透明度
- 动画故事板功能，可以用于所有情形，包括用户触发的事件，如鼠标单击按钮事件
- 可重用的资源，以动态设置控件的样式

许多功能都专门面向 Microsoft Vista 及以后版本的操作系统(包括 Windows 7、Windows Server 2008 和 Windows Server 2008 R2)，这些操作系统可以通过 Aero 接口访问高级图形功能。但是，WPF 应用程序也可以运行在其他操作系统上，例如 Windows XP。如果图形卡因某种原因无法工作，内置于 .NET Framework 4 运行库中的回退渲染系统可以渲染 XAML(但有性能损失)。

VS 和 VCE 包含创建 XAML 代码并设置其样式的功能，但设计人员选择的工具是 Microsoft Expression Blend(EB)。这是一个设计和布局软件包，可用于创建 XAML 文件，接着开发人员就可以使用 XAML 文件创建应用程序。实际上，EB 使用与 VS 和 VCE 相同的解决方案和项目文件，所以可以在这些环境中的一个创建项目，在另一个环境中编辑它。在 EB 中，在编辑代码隐藏文件时，只需在 Files 窗口(等价于 VS 和 VCE 中的 Solution Explorer)中双击它。图 25-2 显示了 EB 界面。

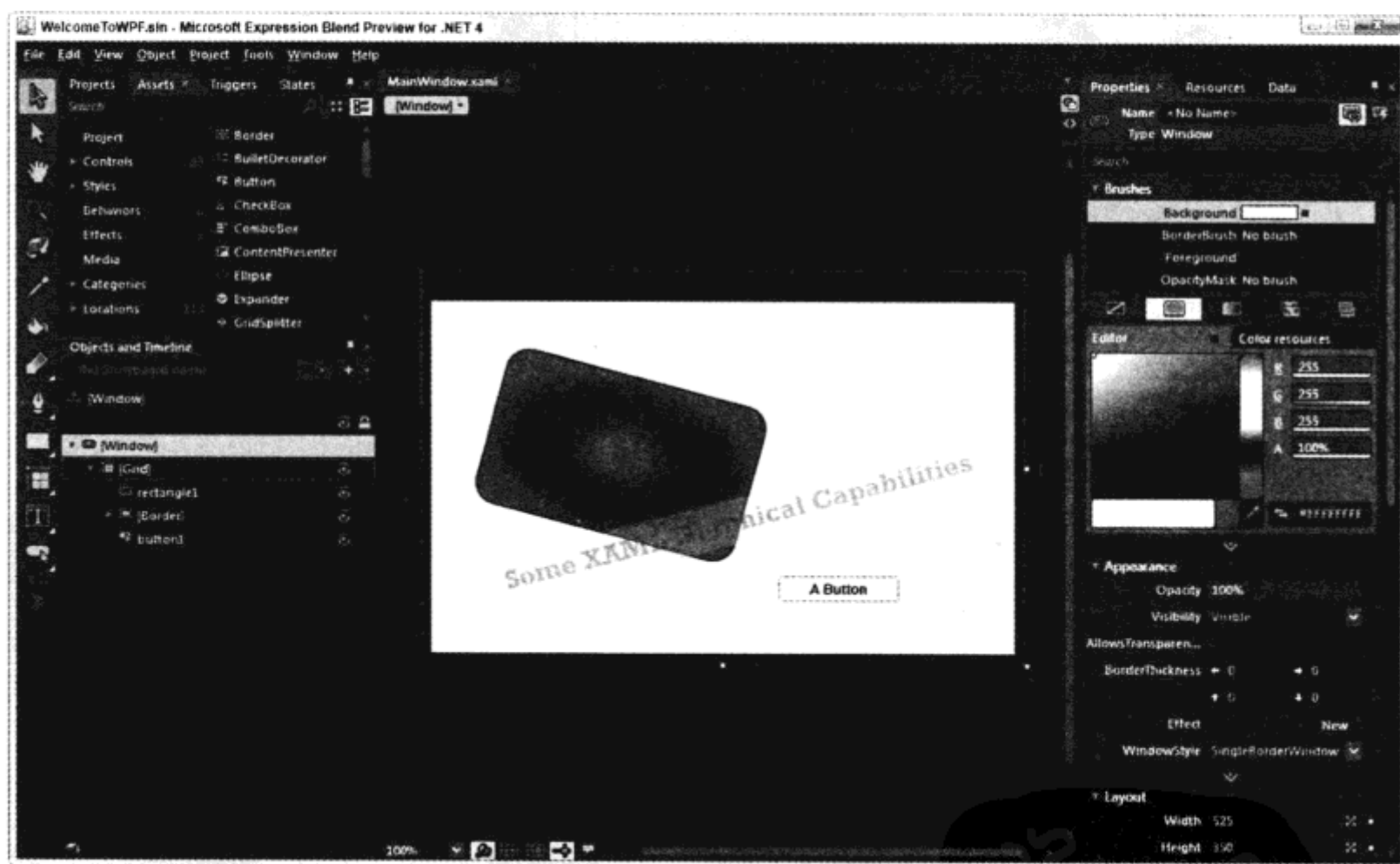


图 25-2

在 [http://microsoft.com/expression/products/Blend\\_Overview.aspx](http://microsoft.com/expression/products/Blend_Overview.aspx) 上可以找到有关 EB 试用版本的许多信息，并可以下载。但是，编写 WPF 应用程序或编辑 XAML 并不需要 EB。图 25-3 显示了图 25-2 中的项目，但该项目在 VCE 中加载。

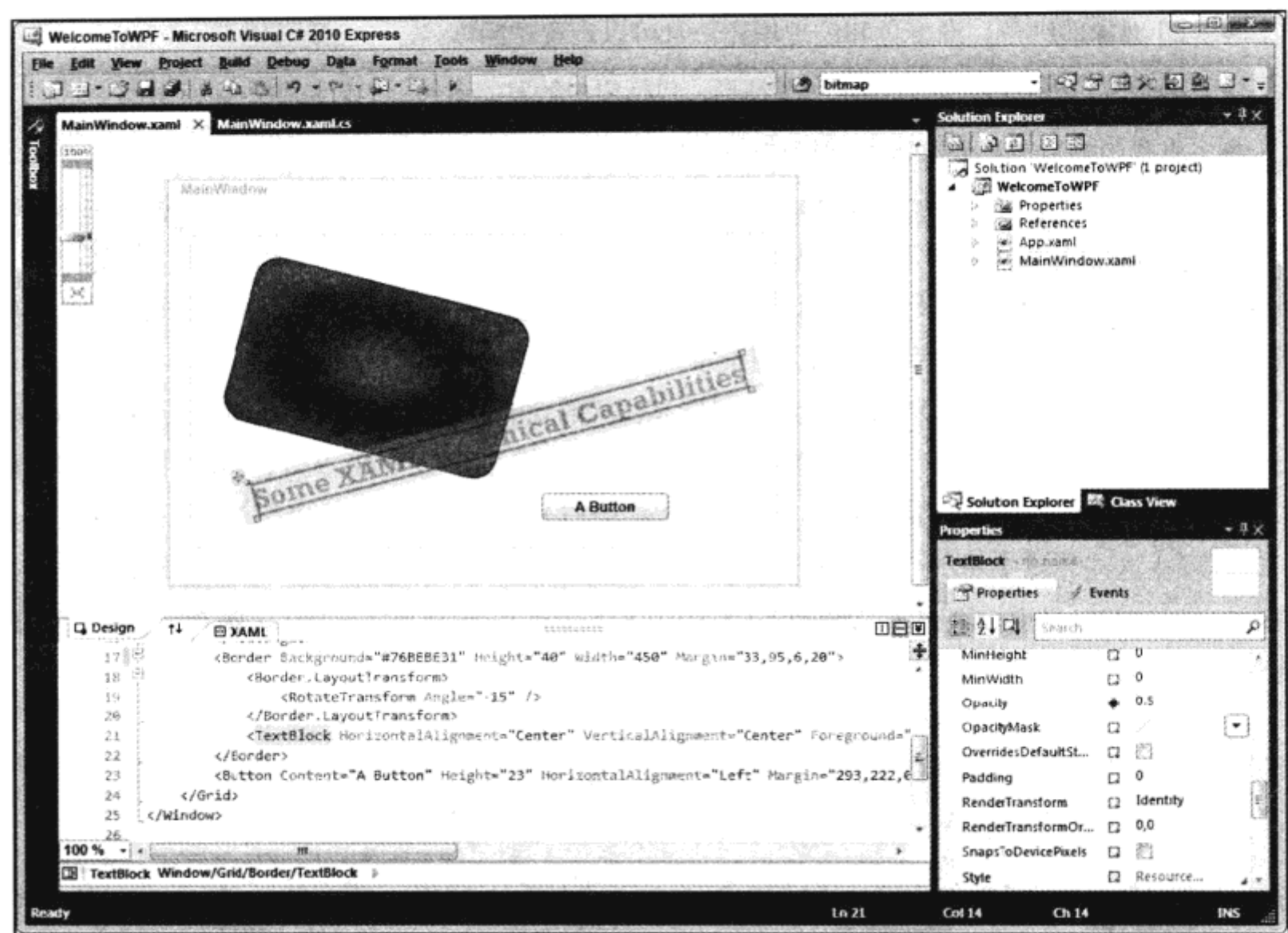


图 25-3

在 VCE 中，默认屏幕会在两个窗格上显示 XAML(目前不必考虑显示在 XAML 视图中的代码)和这些 XAML 的预览。属性编辑器不太直观。  
应用程序在两个环境中启动的效果相同，如图 25-4 所示。

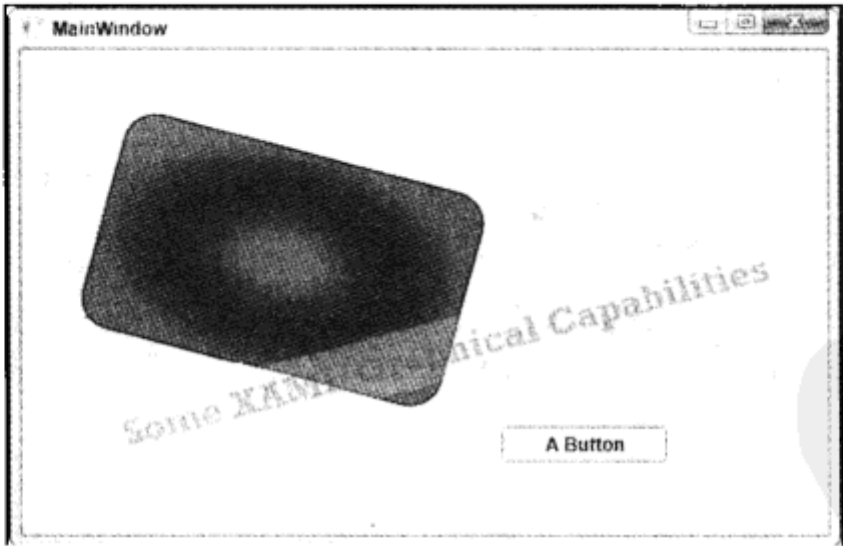


图 25-4

25.1.2 WPF 给 C#开发人员带来的好处

如上一节所述，开发人员可以创建能在 VS 或 VCE 中工作的项目和解决方案，而设计人员可以

在 Expression Blend 中编辑这些项目和解决方案。但与设计人员不同,开发人员主要在 VS 或 VCE 中工作。

如本节的引言所述, WPF 使用与 ASP.NET 非常类似的代码隐藏模式。例如,把 Click 特性添加到表示按钮控件的 XML 元素中,就可以给按钮控件添加一个事件处理程序。这个特性在 XAML 页面的代码隐藏文件中指定了事件处理程序的名称,事件处理程序可以用 C#编写。

注意,还可以在 WPF 应用程序中操作控件,其方式类似于 Windows 窗体应用程序使用编程技术布置用户界面。可以使用代码隐藏实例化控件,设置其属性,添加事件处理程序,给窗口添加控件,这完全绕过了 XAML。其代码一般比对应的 XAML 声明代码长得多,且无法分开设计和功能。编程方式在某些情形下是必须的,但一般应使用 XAML 来布置用户界面上的控件。

本章主要从 C#开发人员的角度来阐述。WPF 是一个需要整本书来介绍的主题,所以本章仅简要探讨它。

## 25.2 基本 WPF 应用程序的组成

WPF 使用起来很直观,学习它的最佳方式是开始使用它。如果读者已经阅读了本书的其他章节,就会很快熟悉许多技术。

下面的示例要创建一个简单的 WPF 应用程序,在该示例的说明中,会详细探讨代码和结果,了解如何把代码组合起来。

### 试一试: 创建一个基本的 WPF 应用程序

- (1) 创建一个新 WPF 应用程序 Ch25Ex01, 将其保存在 C:\BegVCSharp\Chapter25 目录中。
- (2) 修改 MainWindow.xaml 中的代码, 如下所示:



```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex01.MainWindow"
  Title="Color Spinner" Height="370" Width="270">
  <Window.Resources>
    <Storyboard x:Key="Spin">
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse1"
        Storyboard.TargetProperty=
          "(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse2"
        Storyboard.TargetProperty=
          "(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="-360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
```

```

        Storyboard.TargetName="ellipse3"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:05" Value="360"/>
    </DoubleAnimationUsingKeyFrames>
    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse4"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:05" Value="-360"/>
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
</Window.Resources>
<Window.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
        <BeginStoryboard Storyboard="{StaticResource Spin}"
            x:Name="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="goButton">
        <ResumeStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="stopButton">
        <PauseStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
</Window.Triggers>
<Window.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="#FFFFFF" Offset="0"/>
        <GradientStop Color="#FFFC45A" Offset="1"/>
    </LinearGradientBrush>
</Window.Background>
<Grid>
    <Ellipse Margin="50,50,0,0" Name="ellipse5" Stroke="Black" Height="150"
        HorizontalAlignment="Left" VerticalAlignment="Top" Width="150">
        <Ellipse.Effect>
            <BlurEffect Radius="10"/>
        </Ellipse.Effect>
        <Ellipse.Fill>
            <RadialGradientBrush>
                <GradientStop Color="#FF000000" Offset="1"/>
                <GradientStop Color="#FFFFFF" Offset="0.306"/>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Ellipse Margin="15,85,0,0" Name="ellipse1" Stroke="{x:Null}"
        Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
        Width="120" Fill="Red" Opacity="0.5"
        RenderTransformOrigin="0.92,0.5">
        <Ellipse.Effect>
            <BlurEffect/>
        </Ellipse.Effect>
        <Ellipse.RenderTransform>
            <TransformGroup>

```

```

        <RotateTransform Angle="0"/>
    </TransformGroup>
</Ellipse.RenderTransform>
</Ellipse>
<Ellipse Margin="85,15,0,0" Name="ellipse2" Stroke="{x:Null}"
    Height="120" HorizontalAlignment="Left" VerticalAlignment="Top"
    Width="80" Fill="Blue" Opacity="0.5"
    RenderTransformOrigin="0.5,0.92" >
    <Ellipse.Effect>
        <BlurEffect/>
    </Ellipse.Effect>
    <Ellipse.RenderTransform>
        <TransformGroup>
            <RotateTransform Angle="0"/>
        </TransformGroup>
    </Ellipse.RenderTransform>
</Ellipse>
<Ellipse Margin="115,85,0,0" Name="ellipse3" Stroke="{x:Null}"
    Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
    Width="120" Opacity="0.5" Fill="Yellow"
    RenderTransformOrigin="0.08,0.5" >
    <Ellipse.Effect>
        <BlurEffect/>
    </Ellipse.Effect>
    <Ellipse.RenderTransform>
        <TransformGroup>
            <RotateTransform Angle="0"/>
        </TransformGroup>
    </Ellipse.RenderTransform>
</Ellipse>
<Ellipse Margin="85,115,0,0" Name="ellipse4" Stroke="{x:Null}"
    Height="120" HorizontalAlignment="Left" VerticalAlignment="Top"
    Width="80" Opacity="0.5" Fill="Green"
    RenderTransformOrigin="0.5,0.08" >
    <Ellipse.Effect>
        <BlurEffect/>
    </Ellipse.Effect>
    <Ellipse.RenderTransform>
        <TransformGroup>
            <RotateTransform Angle="0"/>
        </TransformGroup>
    </Ellipse.RenderTransform>
</Ellipse>
<Button Height="23" HorizontalAlignment="Left" Margin="20,0,0,56"
    Name="goButton" VerticalAlignment="Bottom" Width="75" Content="Go"/>
<Button Height="23" HorizontalAlignment="Left" Margin="152,0,0,56"
    Name="stopButton" VerticalAlignment="Bottom" Width="75"
    Content="Stop"/>
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
    Name="toggleButton" VerticalAlignment="Bottom" Width="75"
    Content="Toggle"/>
</Grid>
</Window>

```

代码段 Ch25Ex01\MainWindow.xaml



(3) 双击设计视图中的 Toggle 按钮。该按钮在图 25-5 中高亮显示，它会隐藏 XAML 视图。

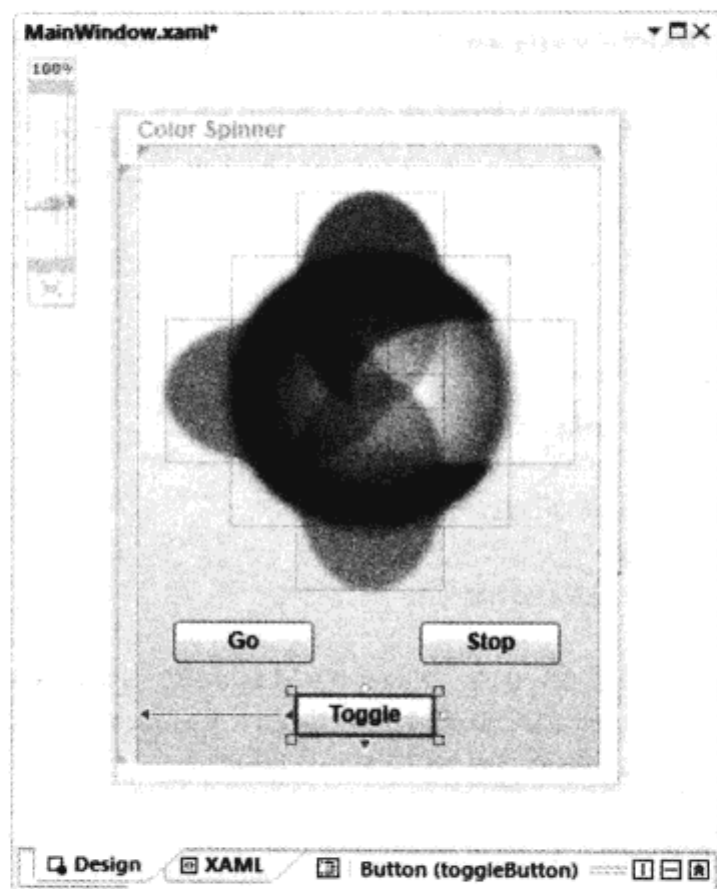


图 25-5

(4) 修改 MainWindow.xaml.cs 中的代码，如下所示(双击按钮时，会在 toggleButton\_Click() 事件处理程序中添加 using 语句和新代码)：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Media.Animation;

namespace Ch25Ex01
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```



```

private void toggleButton_Click(object sender, RoutedEventArgs e)
{
    Storyboard spinStoryboard = Resources["Spin"] as Storyboard;
    if (spinStoryboard != null)
    {
        if (spinStoryboard.GetIsPaused(this))
        {
            spinStoryboard.Resume(this);
        }
        else
        {
            spinStoryboard.Pause(this);
        }
    }
}
}
}

```

代码段 Ch25Ex01\MainWindow.xaml.cs

(5) 执行应用程序，开始、停止、切换动画，示例如图 25-6 所示。

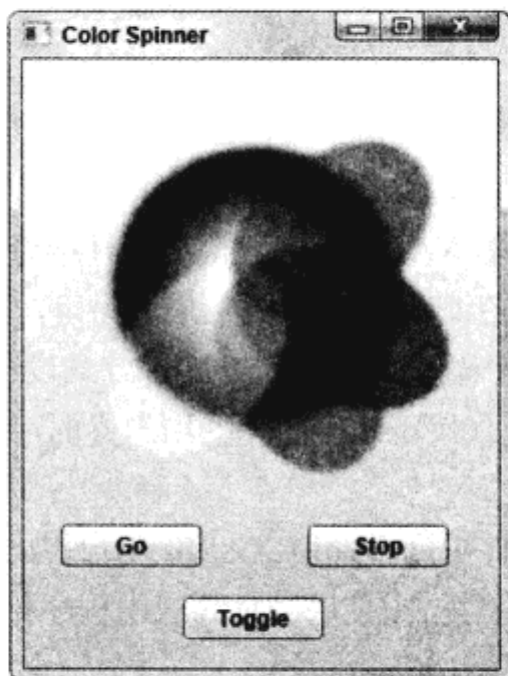


图 25-6

(6) 创建一个新的 WPF 浏览器应用程序 Ch25Ex01Web，将其保存在 C:\BegVCSharp\Chapter25 目录中。

(7) 在 page1.xaml 中把<Page>元素的 Title 特性的值改为 Color Spinner Web。

(8) 打开 Ch25Ex01 应用程序中的 MainWindow.xaml 文件，把该文件中<Window>元素的所有代码复制到 page1.xaml 的<Page>元素中。

(9) 把<Window.Resources>、<Window.Triggers>和<Window.Background>元素分别改为<Page.Resources>、<Page.Triggers>和<Page.Background>元素(注意，要修改这些元素的开始和结束标记)。

(10) 从 page1.xaml 中删除 5 个<Ellipse.Effect>元素及其内容。

(11) 把 toggleButton\_Click() 事件处理程序和 System.Windows.Media.Animation 名称空间的 using

语句从 MainWindow.xaml.cs 复制到 page1.xaml.cs 中。

(12) 执行 Ch25Ex01Web 应用程序，结果如图 25-7 所示。

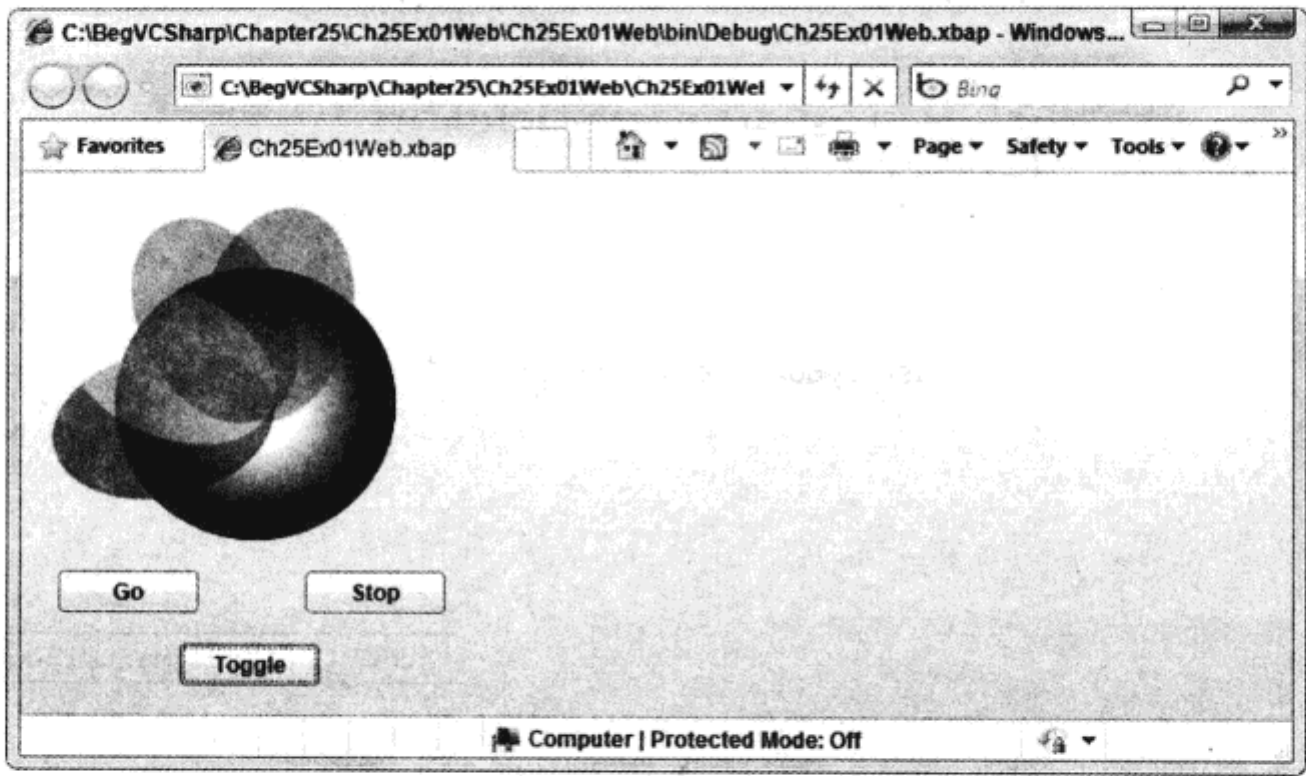


图 25-7

示例的说明

这个示例创建了一个简单应用程序，得到可以开始或停止的彩色旋转椭圆。但黑白屏幕图不能完全传达这种效果，只有运行代码，才会发现应用程序很漂亮。

为了达到这种效果，添加了许多 XAML 代码，但如果仔细查看代码，就会发现许多代码都是重复的——这是必要的，因为要使 4 个椭圆产生动画效果。另外，在代码隐藏文件中几乎没有添加什么 C# 代码，仅为三个按钮中的一个添加了代码。以这种方式设计代码说明了两个要点：

- 设计人员可以仅用 XAML 代码创建出漂亮的用户界面，其中涉及高级图形功能、动画和用户交互操作。
- 需要时可以在代码隐藏文件中完全控制 XAML 用户界面。

我们还介绍了如何在 Web 应用程序中使用与桌面应用程序相同的代码。这需要进行几处修改，如后面所述，但基本功能在两个环境中是相同的。

这个应用程序中的代码演示了 WPF 的许多特性，介绍了一些关键技术。首先创建桌面应用程序，再看看 Web 应用程序需要的修改。下面列出了桌面应用程序 MainWindow.xaml 的 XAML，以及代码的顶级元素：

```
< Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex01.MainWindow"
  Title="Color Spinner" Height="370" Width="270">

  ...

< /Window >
```

<Window>元素用于定义窗口。一个应用程序可以包含几个窗口，每个窗口都包含在一个单独的 XAML 文件中。但这并不是说 XAML 文件总是定义一个窗口。XAML 文件可以包含用户控件、笔刷和其他资源以及 Web 页面等。Ch25Ex01 项目中的 XAML 文件甚至定义了应用程序 App.xaml。本章后面会介绍应用程序和 App.xaml 文件。

在 MainWindow.xaml 中，注意<Window>元素包含一些通过名称即可了解其含义的特性。其中有两个名称空间声明，一个用于全局名称空间(用于 XML)，另一个用于 x 名称空间。这两个名称空间对于 WPF 功能来说非常重要，定义了 XAML 语法的词汇。接着是一个 Class 特性，它取自 x 名称空间。这个特性把 XAML <Window>元素链接到代码隐藏文件中的一个部分类定义上，在这个例子中是 Ch25Ex01.Window。这类似于 ASP.NET，一个类用于一个页面，并允许代码隐藏文件与 XAML 文件共享相同的编码模型，包括 XAML 元素定义的控件等。注意，x:Class 特性只能用于 XAML 文件的根元素。

其他 3 个特性 Title、Height 和 Width 指定了在窗口标题栏中显示的文本以及用于该窗口的尺寸(单位是像素)。这些特性映射到 System.Windows.Window 类的属性上，Ch25Ex01.Window 派生于 System.Windows.Window 类。

System.Windows.Window 类有其他几个属性可以定义额外的功能。许多属性都比<Window>元素上使用的 3 个属性复杂，它们不只是字符串或数字。XAML 语法允许使用嵌套的元素，为这些属性指定值。

“XAML 语法”一节详细讨论了定义对象、属性和内容的 XAML 语法。

例如，下面的代码用一个嵌套的<Window.Background>元素定义了 Background 属性：

```
<Window.Background>
  <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FFFFFF" Offset="0"/>
    <GradientStop Color="#FFFC45A" Offset="1"/>
  </LinearGradientBrush>
</Window.Background>
```

这段代码把 Background 属性设置为 LinearGradientBrush 类的一个实例。在这个示例中，笔刷定义了一个自上而下从白色过渡到桃色的渐变色。

这段代码在嵌套的元素中还定义了另外两个复杂的属性：定义动画的<Window.Resources>和定义触发器的<Window.Triggers>(它控制动画)。这两个属性的功能都远不止此，本章后面会详细介绍它们。

在查看这些属性的实现代码之前，需要先看看<Grid>元素。<Grid>元素定义了 System.Windows.

Controls.Grid 控件的一个实例。这是可以在 WPF 应用程序中用于布局的几个控件之一。它允许用相对于矩形 4 条边的坐标来定位嵌套的控件。其他控件允许用不同的方式定位控件。所有的布局控件都在本章后面的“控件布局”一节中介绍。

<Grid>元素包含 5 个<Ellipse>元素(System.Windows.Shapes.Ellipse 控件)和 3 个<Button>元素(System.Windows.Controls.Button 控件)。这些元素定义的椭圆用于在应用程序中显示旋转图形，按钮用于控制应用程序。

第一个<Ellipse>元素如下所示：

```
<Ellipse Margin="50,50,0,0" Name="ellipse5" Stroke="Black" Height="150"
```

```

HorizontalAlignment="Left" VerticalAlignment="Top" Width="150">
<Ellipse.Effect>
  <BlurEffect Radius="10"/>
</Ellipse.Effect>
<Ellipse.Fill>
  <RadialGradientBrush>
    <GradientStop Color="#FF000000" Offset="1"/>
    <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
  </RadialGradientBrush>
</Ellipse.Fill>
</Ellipse>

```

这个元素定义了 `System.Windows.Shapes.Ellipse` 类的一个实例，用于显示椭圆图形，还设置了这个实例的几个属性，如下所示：

- **Name:** 用于控件的标识符。
- **Margin:** 通过指定图形周围的边距，来指定网格中 `Ellipse` 控件定义的图形的位置。这些数据在代码中以像素为单位。因此，这个属性根据 `HorizontalAlignment` 和 `VerticalAlignment` 属性，映射为图形的实际位置。
- **HorizontalAlignment 和 VerticalAlignment:** 指定使用 `Grid` 定义的矩形的哪一条边布置图形。例如，`Left` 和 `Bottom` 值指定图形相对于网格的左下角定位。
- **Height 和 Width:** 图形的尺寸。
- **Stroke:** `Ellipse` 控件定义的勾画图形轮廓的笔刷。
- **Fill:** `Ellipse` 控件定义的填充图形内部区域的笔刷。
- **Effect:** 显示 `Ellipse` 控件时使用的特效。

`Fill` 属性使用的笔刷是 `RadialGradientBrush`。在这个例子中，笔刷指定了一个从白色(椭圆的中心)过渡到黑色(椭圆的边界)的渐变色。

`Effect` 属性设置为使用 `BlurEffect`。这是可以应用于 WPF 中图形的几个特效之一。这个特效会用 `BlurEffect.Radius` 属性定义的数值模糊图形，它不能应用于 Web 应用程序，因此在第(10)步删除了它。这是桌面应用程序和 Web 应用程序的几个区别之一。



可以定义应用于 XAML 项的定制特效，但这是一个高级主题，本章没有介绍。

代码中的另外 4 个 `<Ellipse>` 元素非常类似。每个元素都定义了四个具有动画效果的彩色椭圆中的一个，第一个元素如下：

```

<Ellipse Margin="15,85,0,0" Name="ellipse1" Stroke="{x:Null}"
  Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="120" Fill="Red" Opacity="0.5"
  RenderTransformOrigin="0.92,0.5" >
  <Ellipse.Effect>
    <BlurEffect/>
  </Ellipse.Effect>
  <Ellipse.RenderTransform>
    <TransformGroup>

```

```

        <RotateTransform Angle="0"/>
    </TransformGroup>
</Ellipse.RenderTransform>
</Ellipse>

```

这段代码非常类似于上一个椭圆的代码，但有如下区别：

- **Stroke** 属性设置为 `{x:null}`。在 XAML 中，放在花括号中的值称为标记扩展，用于为不能在 XAML 语法中简化为简单字符串的属性提供值。在这个例子中，`{x:null}` 为属性指定 `null` 值，表示 **Stroke** 没有使用笔刷。
- **Opacity** 属性指定为 0.5，这表示椭圆是半透明的。
- **Effect** 属性使用 **BlurEffect**，没有指定 **Radius** 特性，这里给 **Radius** 使用默认值 5。
- 指定了 **RenderTransform** 属性。这个属性设置为带一个 **RotateTransform** 变换的 **TransformGroup** 对象。这个变换在椭圆连续变化时使用。它指定了一个属性 **Angle**。椭圆根据这个角度(单位为度)旋转，它最初设置为 0°。
- **RenderTransformOrigin** 用于设置中心点，椭圆在进行 **RotateTransform** 变换时，围绕该中心点旋转。

最后两个属性与 XAML 中定义的动画有关，动画是用 `System.Windows.Media.Animation.Storyboard` 对象定义的。这个对象在 `<Window.Resources>` 元素中定义，表示可以通过窗口的 **Resources** 集合使用 **Storyboard** 对象。这段代码还定义了一个 `x:Key` 特性，它允许使用键通过 **Resources** 引用 **Storyboard** 对象：

```

<Window.Resources>
    <Storyboard x:Key="Spin">
        ...
    </Storyboard>
</Window.Resources>

```

**Storyboard** 对象包含 4 个 **DoubleAnimationUsingKeyFrames** 对象。使用这些对象，可以指定包含 `double` 值的属性应随时间变化，还可以进一步定义这个操作。这段代码中的每个元素都定义了一个彩色椭圆使用的动画。例如，`ellipse1` 椭圆的动画如下所示：

```

<DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
    Storyboard.TargetName="ellipse1"
    Storyboard.TargetProperty=
" (UIElement.RenderTransform).(TransformGroup.Children) [0]
.(RotateTransform.Angle)"
    RepeatBehavior="Forever">
    <SplineDoubleKeyFrame KeyTime="00:00:10" Value="360"/>
</DoubleAnimationUsingKeyFrames>

```

这里不深入探讨这个元素，但它指定了 **RotateTransform** 变换的 **Angle** 属性应在 10 秒内从其初始值变成 360°，而且在变换结束后应重复进行。本章的“动画”一节将详细介绍动画。

定义了椭圆后，有 3 个 `<Button>` 元素定义了按钮(注意 **Click** 特性没有出现在示例的代码中，它是在第(3)步双击按钮时由 IDE 添加的)：

```

<Button Height="23" HorizontalAlignment="Left" Margin="20,0,0,56"
    Name="goButton" VerticalAlignment="Bottom" Width="75"
    Content="Go"/>

```



```

<Button Height="23" HorizontalAlignment="Left" Margin="152,0,0,56"
  Name="stopButton" VerticalAlignment="Bottom" Width="75"
  Content="Stop"/>
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggl" Click="toggleButton_Click"/>

```

这些元素都指定了 `Button` 对象的名称、位置和尺寸，所使用的属性与前面的 `<Ellipse>` 元素的相同。它们还用 `Content` 属性确定了在按钮上显示什么内容——在这个例子中，就是按钮上显示的字符串文本。按钮不仅可以用这种方式显示简单的字符串，还可以使用嵌套的形状或其他图形内容，详见本章的“控件的样式”一节。

`toggleButton` 按钮的 `Click` 特性为 `Click` 事件定义了一个事件处理方法。这个方法即为 `toggleButton_Click()`，它实际上是一个路由的事件(routed event)处理程序。路由的事件详见本章后面的“路由事件”一节。现在只需知道，单击按钮就会触发这个事件，并调用事件处理程序。

在事件处理程序的代码中，首先获得 `Storyboard` 对象的一个引用，该对象定义了动画。如前所述，这个对象包含在 `Window` 对象的 `Resources` 属性中，它使用了键 `Spin`。之后应是检索 `Storyboard` 对象的代码：

```

private void toggleButton_Click(object sender, RoutedEventArgs e)
{
    Storyboard spinStoryboard = Resources["Spin"] as Storyboard;

```

之后，如果前面的代码得到的不是 `null` 值，就使用 `Storyboard.GetIsPaused()` 方法确定动画当前是否暂停。如果是，就调用 `Resume()`，否则就调用 `Pause()`。这两个方法可以恢复和暂停动画的执行：

```

    if (spinStoryboard != null)
    {
        if (spinStoryboard.GetIsPaused(this))
        {
            spinStoryboard.Resume(this);
        }
        else
        {
            spinStoryboard.Pause(this);
        }
    }
}

```

注意，所有这些方法都需要对包含故事板(storyboards)的对象的引用。这是因为故事板本身没有跟踪时间。包含故事板的窗口有自己的时钟，故事板就使用这个时钟。给故事板传送对窗口的引用(通过 `this`)，故事板就可以访问这个时钟。

另外两个按钮 `goButton` 和 `stopButton` 没有链接到后台代码的任何事件处理方法上。它们的功能是由触发器决定的。在这个例子中，定义了 3 个触发器，如下所示：

```

<Window.Triggers>
  <EventTrigger RoutedEvent="FrameworkElement.Loaded">
    <BeginStoryboard Storyboard="{StaticResource Spin}"
      x:Name="Spin_BeginStoryboard"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="ButtonBase.Click"

```

```

        SourceName="goButton">
        <ResumeStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click"
        SourceName="stopButton">
        <PauseStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
</Window.Triggers>

```

第一个触发器将 `FrameworkElement.Loaded` 事件(在加载应用程序时启动)和 `BeginStoryboard` 动作链接起来。这个动作启动 `Spin` 动画。注意, `Spin` 动画是通过代码(`StaticResource Spin`)使用标记扩展语法引用的。这个语法用于引用包含窗口中的资源,它在 WPF 应用程序中很常见。`BeginStoryboard` 动作的名称是 `Spin_BeginStoryboard`,在另外两个触发器中引用,这两个触发器分别链接了 `goButton` 和 `stopButton` 的 `Click` 事件。这些触发器使用了 `ResumeStoryboard` 和 `PauseStoryboard` 动作,恢复和暂停动画的执行。

这段代码在作为桌面应用程序时工作正常,但把它们转换为 Web 应用程序时,需要进行几处修改。实际上,这个示例创建了一个新的 Web 浏览器应用程序,隐藏了几个细节。例如,在浏览器上运行代码有一些安全方面的限制,所以 Web 浏览器应用程序使用了一个临时的键,它用于给应用程序签名。如果希望让应用程序执行浏览器应用程序禁止执行的某些动作,例如,访问本地文件系统,使用临时键就是必须的。

还要注意,桌面应用程序的根元素 `<Window>` 在 Web 应用程序中被 `<Page>` 元素替换。这是因为浏览器提供的功能略微不同于运行桌面应用程序的主机程序。因此, WPF 使用不同的类表示这些不同的主机。但是如代码所示,许多对象都可以在两个环境中使用相同的代码,详见本章后面的内容。

这就完成了这个示例应用程序的分析。这里复习了许多基础知识,还简要学习了一些新概念,所以现在最好休息一下。本章剩余的内容将深入阐述这里提到的技术,规范需要的语法,再介绍一些新知识。

## 25.3 WPF 基础

希望本章第一部分的示例能激起读者对 WPF 编程的兴趣。虽然还有许多新概念需要理解,但我们已经介绍了如何组合 XAML 与 .NET 代码快速创建动态的应用程序。还学习了许多功能,包括设计人员不需要 C# 的任何知识,就可以设计应用程序的 UI。最后说明了如何用几乎相同的代码创建桌面应用程序和 Web 应用程序。

但是,在开始创建 WPF 应用程序之前,应花点时间学习基础知识。本节就介绍 WPF 应用程序的几个基础主题,学习实现它们所需的语法。还要探讨许多可以在应用程序中进一步研究的其他方法。

本节的内容如下:

- XAML 语法
- 桌面应用程序和 Web 应用程序
- Application 对象
- 控件基础知识,包括依赖属性、关联属性、路由事件和关联事件



- 控件的布局和样式
- 触发器
- 动画
- 静态和动态资源

### 25.3.1 XAML 语法

本章第一部分的示例虽然介绍了许多 XAML 语法,但没有正式描述它们。这个示例忽略了许多规则和可能性,以便只考虑基本结构和功能。本节将详细介绍 XAML,理解 XAML 文件的组成部分。

#### 1. 对象元素语法

XAML 文件的基本结构使用对象元素语法来描述对象的一个层次结构,这个层次结构有一个根对象,它包含了其他所有对象。顾名思义,对象元素语法描述了用 XML 元素表示的对象(或结构)。例如,在前面的示例中,<Button>元素用于表示 System.Windows.Controls.Button 对象。

XAML 文件的根元素总是使用对象元素语法,但在前面的示例中,用于根对象的类不是用元素名(<Window>或<Page>)定义的,而是用 x:Class 特性定义的。这个语法只能用于根元素。对于桌面应用程序,根元素必须继承于 System.Windows.Window,而对于 Web 应用程序,根元素必须继承于 System.Windows.Controls.Page。

用对象元素语法定义的许多对象实际上都是控件,如前面示例中使用的 Button 控件。

#### 2. 特性语法

在许多情况下,元素用于表示对象(使用对象元素语法)时,就用特性指定属性和事件。例如,前面的<Button>元素使用了如下特性:

```
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
    Name="toggleButton" VerticalAlignment="Bottom" Width="75"
    Content="Toggle" Click="toggleButton_Click"/>
```

这里的每个特性都设置了 toggleButton 对象的一个属性值,但 Click 和 Name 除外,Click 特性把一个路由的事件处理程序赋予 toggleButton 的 Click 事件。这些都是特性语法的例子。

这里使用的 Name 特性是一个特例:它定义了控件的标识符,以便在代码隐藏文件和其他 XAML 代码中引用它。

特性可以用它们引用的基类和一个句点来限定。例如,Button 控件从 ButtonBase 中继承了 Click 事件,所以可以把上面的代码重写为:

```
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
    Name="toggleButton" VerticalAlignment="Bottom" Width="75"
    Content="Toggle" ButtonBase.Click="toggleButton_Click"/ >
```

注意,这个语法也称为关联属性,参见本章的“控件基础”一节。

#### 3. 属性元素语法

在许多情况下,需要使用比简单字符串略微复杂的方式来初始化属性值。在示例应用程序中,

Fill 属性就是这样，它设置为各种笔刷对象：

```
<Ellipse ...>
...
<Ellipse.Fill>
  <RadialGradientBrush>
    <GradientStop Color="#FF000000" Offset="1"/>
    <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
  </RadialGradientBrush>
</Ellipse.Fill>
</Ellipse>
```

这里属性通过子元素设置，该子元素的名称根据下面的约定来指定：

```
[Parent Element Name].[Property Name]
```

这称为属性元素语法。

#### 4. 内容语法

许多控件其实都是内容表示器(content presenter)。这表示可以给控件提供内容，这些内容根据控件模板显示出来。例如，为 Button 控件提供的内容显示在按钮表面。这些内容可以是文本，如示例所示，也可以是图形。

使用内容语法可以为控件指定内容，为此，只需把内容添加为表示控件的元素的内容：

```
<Button ...>Go</Button>
```

这行代码用于 Button 控件，以显示文本 Go。这个示例使用了一种比较简单但不太灵活的方式：特性 Content。这里使用的完整内容语法对于较复杂的内容而言是必须的，例如，本节引言中提到的图形内容。

给控件使用复杂的内容时，XAML 代码会有复杂的嵌套层次。因此，XAML 允许控件用其他更微妙的方式设置样式。有关内容表示器和样式设置的知识详见“控件的样式”一节。

#### 5. 合并属性元素语法和内容语法

XAML 页面上用对象元素语法格式化的控件可能包括属性元素语法和内容语法。此时，必须遵循下述语法规则：

- 用于属性元素语法的元素不必是连续的，即一个使用属性元素语法的元素后面可以跟一个使用内容语法的元素，其后的第 3 个元素再使用属性元素语法。
- 用于内容语法的元素(和文本内容)必须是连续的——使用内容语法的文本或元素后不能跟一个使用属性元素语法的元素，其后的第 3 个元素再使用文本或内容语法。

因此下面的代码是正确的：

```
<Button ...>
  <Button.Effect>
    <BlurEffect Radius="10"/>
  </Button.Effect>
  Go
  <Button.RenderTransform>
    <RotateTransform Angle="20"/>
  </Button.RenderTransform>
</Button>
```

```
</Button.RenderTransform>
</Button>
```

但是，下面的代码不能工作，因为有两个地方使用了内容语法，它们用一个使用属性元素语法的元素隔开了：

```
<Button ...>
    Don't
    <Button.Effect>
        <BlurEffect Radius="10"/>
    </Button.Effect>
    Go
</Button>
```

6. 标记扩展

前面的示例说明了标记扩展也可以用于属性值——例如，值{x:null}。只要使用了花括号{}，就是在使用标记扩展。这些标记扩展都可以在特性语法和属性元素语法代码中使用。

本章的所有标记扩展都专门用于 WPF。WPF 特有的扩展包括用于引用资源和数据绑定的扩展。

25.3.2 桌面和 Web 应用程序

本章前面的示例演示了 WPF 应用程序如何作为独立的桌面应用程序和 Web 应用程序运行。前面把 WPF Application 和 WPF Browser Application 项目模板作为起点，添加了 XAML 和 C#代码，以完成该应用程序。WPF Application 模板把项目编译为.exe 文件，WPF Browser Application 模板把项目编译为.xbap 文件。



XBAP(读作 ex-bap)是 XAML Browser Application 的首字母的缩写，用 WPF 创建的 Web 应用程序常称为 XBAP 应用程序。

这些应用程序类型之间的大多数区别都是项目文件(.csproj)之间的区别。Web 浏览器应用程序定义了一些额外的设置，包括两个应用程序的签名和应用程序的清单(如前所述，为了安全起见)，以及允许在浏览器中调试的设置。还有一个测试证书，可用于这个签名。在生产环境下，需要用证书颁发机构颁发的证书替代这个证书。

在桌面 WPF 和 Web WPF 应用程序之间转换是一个难度很高的过程，因为必须改变许多设置，还要修改一些 XAML 代码，如示例所示。这些改动包括把<Window>元素改为<Page>元素，删除位图效果等功能。最好的方法通常是创建独立的项目，如前面的示例所示。

25.3.3 Application 对象

在 WPF 中，大多数应用程序(包括所有的 XBAP 应用程序和使用 WPF Application 模板的桌面应用程序)都包含一个派生自 System.Windows.Application 的类实例。在前面的示例应用程序中，这个对象由 App.xaml 和 App.xaml.cs 文件定义。Ch25Ex01 的 App.xaml 如下所示：



可从  
wrox.com  
下载源代码

```
<Application x:Class="Ch25Ex01.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

```
StartupUri="MainWindow.xaml">
<Application.Resources>

</Application.Resources>
</Application >
```

代码段 Ch25Ex01\App.xaml

<Application>元素的语法类似于前面讨论的<Window>元素，它以相同的方式使用 x:Class 特性，把代码链接到代码隐藏文件中的部分类定义上。

这段代码定义的对象是 WPF 应用程序的入口。这个对象只能有一个实例，使用静态属性 Application.Current 可以通过代码访问它。应用程序的 Application 对象非常有用，原因如下：

- 它提供了在应用程序的生命周期的特定时刻引发的许多事件，包括前面介绍的 LoadCompleted 和 DispatcherUnhandledException，LoadCompleted 在应用程序加载并显示时引发，DispatcherUnhandledException 在抛出一个未处理的异常时引发。
- 它包含的方法可以用于设置或加载 cookie，定位和加载资源等。
- 它的几个属性可用于访问应用程序范围内的资源(参见“静态和动态资源”一节)和应用程序中的窗口。

Application 对象引发的事件是这个列表中最有用的特性，也是最常用的特性。

25.3.4 控件基础

WPF 提供了许多可用于创建应用程序的控件。本章简要介绍 WPF 及其功能，所以不详细探讨每个 WPF 控件，而是通过使用它们来学习。

下面列出了 WPF 提供的控件：

Border	Image	Slider
BulletDecorator	Label	StatusBar
Button	ListBox	TabControl
Calendar	ListView	TextBlock
CheckBox	ListView	TextBox
ComboBox	Menu	ToolBar
ContextMenu	MediaElement	ToolTip
DataGrid	PasswordBox	TreeView
DatePicker	Popup	Viewbox
DocumentViewer	ProgressBar	
Expander	RadioButton	
FlowDocumentPageViewer	RepeatButton	
FlowDocumentReader	RichTextBox	
FlowDocumentScrollViewer	ScrollBar	
Frame	ScrollViewer	
GroupBox	Separator	



这个列表不包含用于布局的 WPF 控件，这些控件参见本章后面的内容。

这里列出的一些控件的名称您或许非常熟悉，实际上，它们的功能与 Windows 窗体和 ASP.NET 应用程序中的对应控件非常类似。例如，**Button** 控件可用于显示按钮。其他一些控件您可能不太熟悉，所以需要试用，以了解它们的功能。

这些控件最初都只有非常基本的外观。为了美化它们，必须给它们设置样式，这会使 WPF 的强大功能变得非常明显，如本章后面所述。除了设置样式之外，WPF 控件还使用了其他几个特性。本节将介绍如下内容：

- 依赖属性
- 关联属性
- 路由事件

与其他桌面应用程序和 Web 应用程序开发一样，也可以创建自己的控件，而且您几乎肯定会创建这种控件。在创建控件时，可以使用这里介绍的所有特性。下面几节会举几个实现例子。

### 1. 依赖属性

依赖属性(dependency property)是在整个 WPF 中使用的一种属性，尤其是在控件上使用，它提供了扩展一般.NET 属性的功能。为了说明这一点，考虑某个一般的.NET 属性。在.NET 中创建类时，一般使用非常简单的代码来实现属性：

```
private string aStringProperty;

public string AStringProperty
{
    get
    {
        return aStringProperty;
    }
    set
    {
        aStringProperty = value;
    }
}
```

这里定义了一个公共属性，它以一个私有字段为基础。这些简单的实现代码绝对适用于大多数目的，但除了基本的状态访问之外，没有包含太多的功能。例如，如果要给控件 **ControlA** 添加一个 **AStringProperty**，让另一个控件 **ControlB** 响应对该属性的改动，就必须执行如下步骤：

- (1) 使用前面的代码给控件 **ControlA** 添加 **AStringProperty**。
- (2) 给 **ControlA** 添加一个事件。
- (3) 给 **ControlA** 添加一个方法，来引发事件。
- (4) 在 **ControlA** 中为 **AStringProperty** 的 **set** 访问器添加代码，以调用事件，引发方法。
- (5) 给 **ControlB** 添加代码，以订阅 **ControlA** 中的事件。





这里不需要列出添加和响应简单属性的变化的代码，因为本书已经多次列出了这些代码。

这种方法存在的问题是要达到这个效果，没有可以遵循的明确标准。不同开发人员可能采用不同方式添加代码，来达到相同的效果。而且，这需要在开发控件时确定所有可能需要通知他人的属性。

这个问题的WPF 解决方案是用依赖属性替代前面代码中使用的简单属性定义，然后使用格式化的、结构化的技术提供属性改变通知。依赖属性用WPF 属性系统注册，允许使用扩展的功能。这个扩展功能包括(但不仅限于)属性改动的自动通知。确切地讲，依赖属性有如下特性：

- 可以使用样式改变依赖属性的值。
- 可以使用资源或通过数据绑定设置依赖属性的值。
- 可以改变动画中依赖属性的值。
- 可以在 XAML 中按层次设置依赖属性——在父元素上设置的依赖属性值可以用于设置其子元素的对应依赖属性的默认值。
- 使用定义好的编码模式可以配置属性值改动的通知。
- 可以配置一系列相关属性，在改变其中一个属性值时，它们就会全部更新。这称为强制转换。改变的属性会强制转换其他属性的值。
- 可以把元数据应用于依赖属性，指定其他行为特征。例如，可以指定如果给定属性改变了，就需要重新布置用户界面。

实际上，由于依赖属性的实现方式，我们可能最初注意不到它们与一般属性有什么区别。但是，在创建自己的控件时，很快会发现如果使用一般的.NET 属性，有许多功能都突然消失了。

依赖属性在 WPF 中使用得非常普遍，所以后面将介绍如何实现它们。

## 2. 关联属性

关联属性是一种属性，定义这种属性的类的实例的每个子对象都可以访问它。例如，假定有一个类 `Recipe`，它可以包含表示成分的子对象。在 `Recipe` 类定义中可以定义一个关联属性 `Quantity`，它可以由每个子对象使用。注意，子对象不必为关联属性指定值。

这么做的主要原因是用于设置关联属性值的 XAML 代码很容易理解：

```
<Recipe Name="Simple Vegetable Chili">
  <TinOfKidneyBeans Recipe.Quantity="2" Mashed="true" />
  <TinOfChoppedTomatoes Recipe.Quantity="2" />
  <FreshChili Recipe.Quantity="5" Notes="Chopped fine, vary to taste." />
  <Onion Recipe.Quantity="1" Notes="Chopped and fried in olive oil." />
  <LBVPort Notes="Just a dash." />
</Recipe>
```

这里使用的语法是前面介绍的特性语法的一种形式。其中关联属性使用父元素名、句点和关联属性名来表示。

在 WPF 中, 关联属性有许多用途。稍后在“控件的布局”一节中介绍如何定位控件时, 会介绍许多关联属性。我们将学习容器控件如何定义关联属性, 让每个子控件都可以确定(例如)要停靠在容器的哪条边上。

### 3. 路由事件

WPF 应用程序的本质是层次结构, 所以其中的控件常常包含其他控件, 这些控件又包含更多的控件, 依此类推。路由事件(routed event)是一个机制, 借助这个机制, 影响层次结构中一个控件的事件可以影响层次结构中的其他事件, 且不需要复杂的代码。

一个很好的例子是允许用户用鼠标与应用程序交互操作, 当然这非常常见。用户单击应用程序中的按钮时, 一般要响应单击事件。Windows 窗体和 ASP.NET 开发中的一种常见方式是为按钮的事件提供事件处理程序, 来响应鼠标的单击。

这种技术存在很大的局限性, 且在一些 Windows 窗体应用程序中会导致代码比较混乱, 尽管初看起来不是这样。其原因是需要某种机制引发某个按钮的单击事件, 标识该控件应响应鼠标单击, 而这并不是非常明显。在这里举出的简单例子中, 是应引发按钮的单击事件, 还是应引发包含按钮的窗口的单击事件? 如果按钮和窗口都有事件处理程序, 且只引发了其中一个事件, 一般希望引发按钮的事件。但如果希望引发两个事件, 那么引发事件的顺序如何? 对于 Windows 窗体应用程序, 这需要编写相当复杂的定制代码。

WPF 控件(包括 Button 和 Window)的鼠标单击事件实现为路由事件, 解决了这个问题。路由事件由层次结构中的所有对象按指定顺序引发, 可以完全控制响应它们的方式。

例如, 假定 Window 包含一个 Grid, Grid 又包含一个 Rectangle。单击 Rectangle 时, 事件的引发顺序如下:

- (1) 引发 Window 上的鼠标按下事件
- (2) 引发 Grid 上的鼠标按下事件
- (3) 引发 Rectangle 上的鼠标按下事件
- (4) 引发 Rectangle 上的另一个鼠标按下事件
- (5) 引发 Grid 上的另一个鼠标按下事件
- (6) 引发 Window 上的另一个鼠标按下事件。

添加适当的事件处理方法, 就可以响应上述序列中的任一个事件。还可以在事件处理方法的任一点中断该序列, 但事件处理程序默认不会中断该序列。这说明, 可以在一个事件(这里是一个鼠标按下事件)中触发多个事件处理方法。

在描述前面的事件序列时, WPF 引入了一些有用的术语。事件在控件的层次结构中向下移动时, 称为通道(tunneling); 向上移动时, 称为冒泡(bubbling)。

另外, 只要在 WPF 中使用了路由事件, 事件名就可以指出该事件是通道事件还是冒泡事件。所有通道事件都以前缀 Preview 开头。例如, Window 控件有 PreviewMouseDown 和 MouseDown 事件。可以给它们中的一个或两个添加处理程序, 或者不添加任何处理程序。

图 25-8 列出了上述事件的引发顺序, 并说明了事件在控件层次结构中如何通过通道和冒泡。



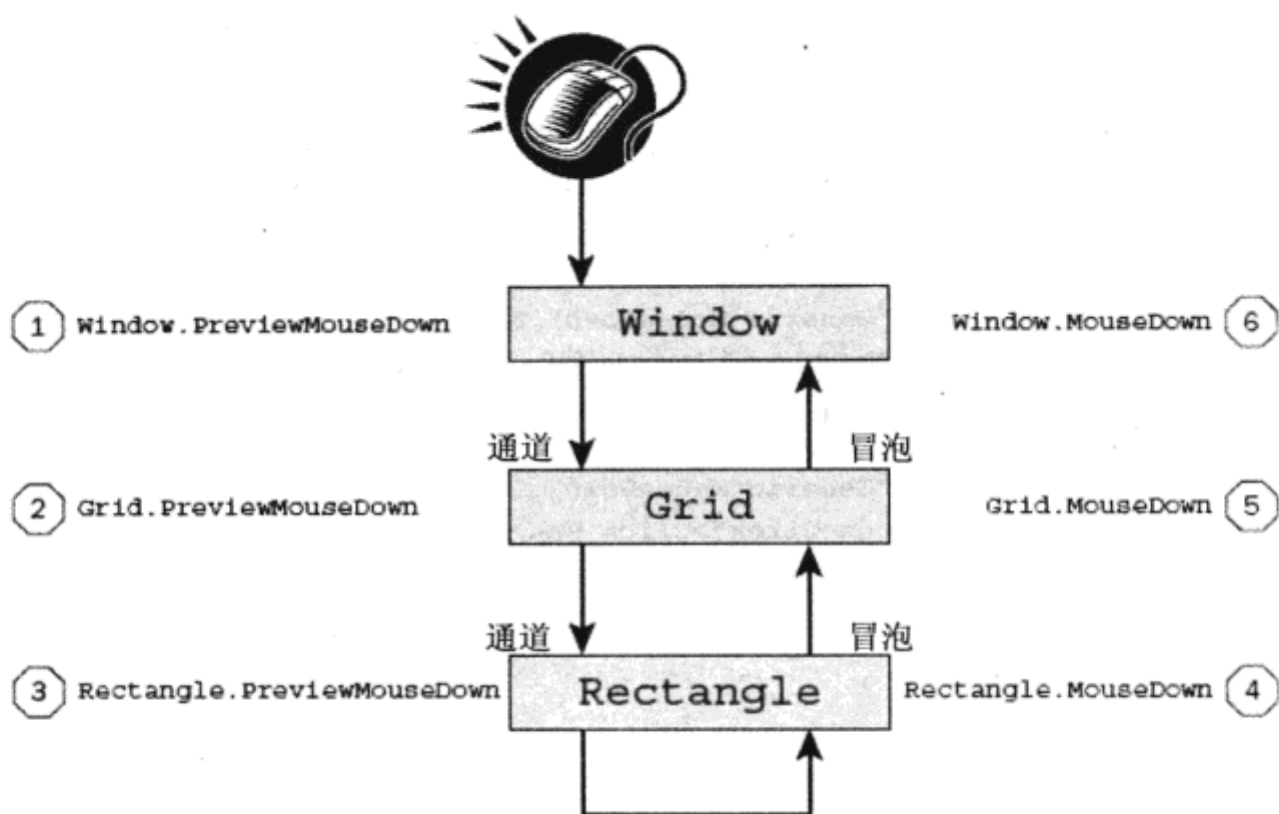



图 25-8

路由事件处理程序有两个参数：事件源和 `RoutedEventArgs` 实例或派生于 `RoutedEventArgs` 的类。实现路由事件的事件处理程序时，可以根据需要将 `RoutedEventArgs` 对象的 `Handled` 属性设置为 `true`。这样，就不会执行进一步的处理了，即该事件不会引发更多的事件处理程序。

`RoutedEventArgs` 还有一个属性 `Source`，它允许指定哪个控件最先引发了事件。这是 WPF 最初检测到事件的控件，所以在图 25-8 中，该控件应是 `Rectangle`。这是非常有用的，因为父控件可以确定单击了哪个子控件。注意，这个“单击测试”是相当复杂的，例如，WPF 可以忽略控件透明区域上的单击，我们不需要做任何工作，就可以启用这个功能。另外，还可以创建可响应鼠标单击的透明控件，所以非常灵活。




WPF 在遇到单击测试时，会区分控件的“透明”区域和“空”区域。只有“透明”区域会响应单击测试，“空”区域会被忽略。

路由事件覆盖了多个鼠标单击，它们可以用于各种目的，包括键盘交互操作、数据绑定和计时器等。稍后介绍的关联事件会使路由事件更有用。

下面的示例演示了本节描述的情形，还介绍了路由事件的其他信息。

试一试：处理路由事件

- (1) 创建一个新 WPF 应用程序 Ch25Ex02，将其保存在 C:\BegVCSharp\Chapter25 目录中。
- (2) 修改 `MainWindow.xaml` 的代码，如下所示：



可从  
wrox.com  
下载源代码

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex02.MainWindow"
  Title="Routed Events" Height="400" Width="800"
```

```

MouseDown="Generic_MouseDown" PreviewMouseDown="Generic_MouseDown"
MouseUp="Window_MouseUp">
<Grid Name="contentGrid" MouseDown="Generic_MouseDown"
  PreviewMouseDown="Generic_MouseDown" Background="Azure">
  <Rectangle Name="clickMeRectangle" Margin="10,10,0,0"
    Height="23" HorizontalAlignment="Left" VerticalAlignment="Top"
    Width="70" Stroke="Black" MouseDown="Generic_MouseDown"
    PreviewMouseDown="Generic_MouseDown" Fill="CadetBlue" />
  <Button Name="clickMeButton" Margin="0,10,10,0" Height="23"
    HorizontalAlignment="Right" VerticalAlignment="Top" Width="70"
    MouseDown="Generic_MouseDown"
    PreviewMouseDown="Generic_MouseDown"
    Click="clickMeButton_Click">Click Me</Button>
  <TextBlock Name="outputText" Margin="10,40,10,10"
    Background="Cornsilk" />
</Grid>
</Window>

```

代码段 Ch25Ex02\MainWindow.xaml

(3) 修改 MainWindow.xaml.cs 中的代码, 如下所示(注意, 根据所使用的 IDE 和输入 XAML 代码的方式, 可能会自动添加空事件处理方法):



可从  
wtox.com  
下载源代码

```

public partial class MainWindow : Window
{
    ...

    private void Generic_MouseDown(object sender,
        MouseButtonEventArgs e)
    {
        outputText.Text = string.Format(
            "{0}Event {1} raised by control {2}. e.Source={3}\n",
            outputText.Text,
            e.RoutedEvent.Name,
            sender.ToString(),
            ((FrameworkElement)e.Source).Name);
    }

    private void Window_MouseUp(object sender, MouseButtonEventArgs e)
    {
        outputText.Text = string.Format(
            "{0}=====\n\n",
            outputText.Text);
    }

    private void clickMeButton_Click(object sender, RoutedEventArgs e)
    {
        outputText.Text = string.Format(
            "{0}Button clicked!\n=====\n\n",
            outputText.Text);
    }
}

```

代码段 Ch25Ex02\MainWindow.xaml.cs

(4) 运行应用程序。当程序运行时，依次单击左上角的矩形、矩形和按钮之间的浅蓝色区域，以及按钮，结果如图 25-9 所示。

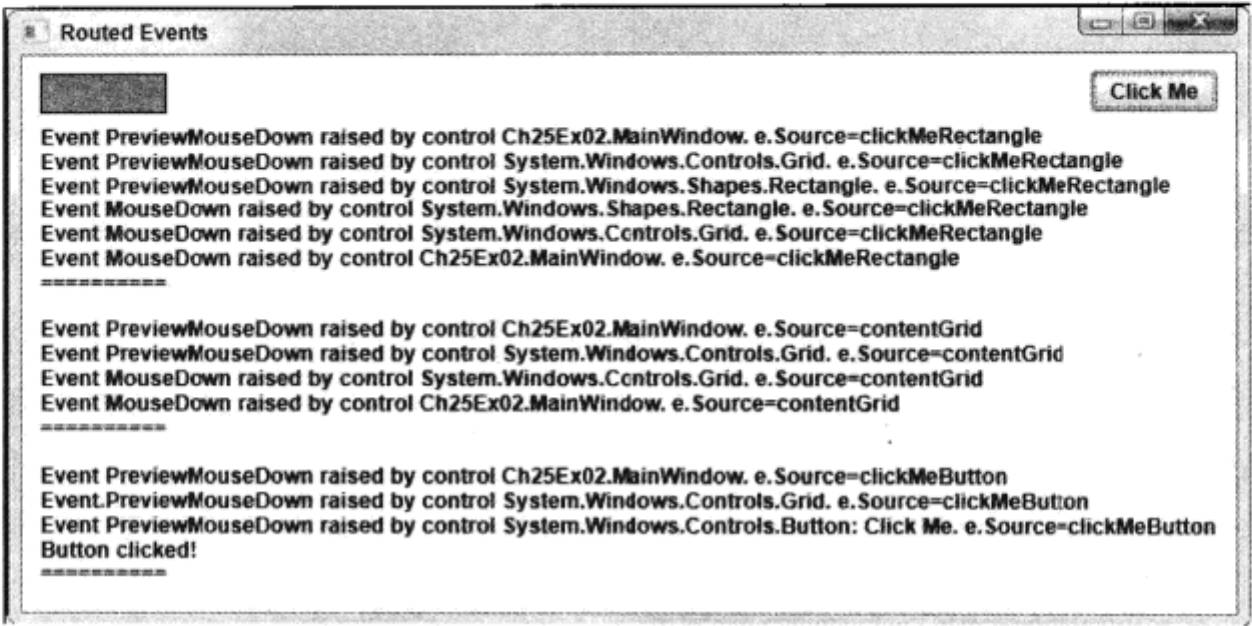


图 25-9

示例的说明

这个示例突出显示了所有 WPF 控件都有的 PreviewMouseDown 和 MouseDown 事件，说明了如何处理路由事件。还介绍了在事件链中包含按钮时会发生什么。所使用的 XAML 代码非常简单，其基本部分(在这个示例中)如下：

```
<Window
  x:Class="Ch25Ex02.MainWindow" MouseDown="Generic_MouseDown"
  PreviewMouseDown="Generic_MouseDown" MouseUp="Window_MouseUp">
  <Grid Name="contentGrid" MouseDown="Generic_MouseDown"
    PreviewMouseDown="Generic_MouseDown">
    <Rectangle Name="clickMeRectangle" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown" />
    <Button Name="clickMeButton" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown"
      Click="clickMeButton_Click" />
    <TextBlock Name="outputText" />
  </Grid>
</Window>
```

这里所有对性能没有影响的属性都被删除了，以便主要考虑与路由事件处理相关的代码。本例使用了 3 个事件处理程序，其配置如表 25-1 中所示。

表 25-1

事件处理程序	处理的事件
Generic_MouseDown()	Window.PreviewMouseDown
	Window.MouseDown
	Grid.PreviewMouseDown
	Grid.MouseDown
	Rectangle.PreviewMouseDown
	Rectangle.MouseDown

(续表)

事件处理程序	处理的事件
Window_MouseUp()	Window.MouseUp
clickMeButton_Click()	Button.Click

事件处理方法只是把信息输出到 TextBlock 控件中，说明发生了什么。文本输出包括事件名、引发事件的控件和事件的源控件(通过 RoutedEventArgs.Source 获得)。

运行应用程序时，第一次单击 Rectangle 控件会发生示例前面描述的事件序列。Generic\_MouseDown()事件处理程序会调用 6 次：其中 3 次为 PreviewMouseDown 通道事件调用，另外 3 次为 MouseDown 冒泡事件调用。所有这些事件的事件源都是满足单击测试的控件，即矩形 clickMeRectangle。在这些处理程序之后还调用了 Window\_MouseUp()事件处理程序，添加了一些文本，把这个测试与下一次测试分开。

此后单击控件之间的区域，实际上这会单击 Grid 控件的背景。这次调用了 4 次 Generic\_MouseDown()事件处理程序：其中 2 次为 PreviewMouseDown 通道事件调用，另外 2 次为 MouseDown 冒泡事件调用。在所有情况下，事件源是 Grid 控件 contentGrid。在调用 Generic\_MouseDown()后，又调用了 Window\_MouseUp()事件处理程序。

最后单击 Button 控件。这次 Generic\_MouseDown()仅调用了 3 次，接着触发 Button.Click 事件，调用 clickMeButton\_Click()。然后调用 Window\_MouseUp()事件处理程序。

在最后这个事件链中，MouseDown 事件由按钮处理，用于触发其 Click 事件。按钮的 MouseDown 事件处理程序的底层实现代码把 RoutedEventArgs 事件参数的 Handled 属性设置为 true。这会中断事件流，所以 MouseDown 事件不会按层次结构向上传送。

为了弄明白按钮控件中断事件路由的方式，我们在介绍示例前单击了 Rectangle 控件。

4. 关联事件

前面的示例在页面的 Button 控件上添加了 Button.Click 事件。但没有给 Grid 或 Window 的 Click 事件添加处理程序，因为这两个控件没有 Click 事件。但有时我们希望它们有 Click 事件。

例如，假定一个窗口包含 1000 个按钮，而您希望处理每个按钮的 Click 事件。这需要 1000 个事件处理程序，或者简化编码，使用一个共享的事件处理程序。但即使只有一个事件处理程序，也必须把它关联到每个 Button.Click 事件上。

WPF 为处理这种情况提供了另一种更好的方式：关联事件。使用关联事件系统，可以在没有提供事件的控件上处理这些事件，所以上面的示例可以处理包含按钮的 Grid 上的 Button.Click 事件——即使 Grid 控件没有 Click 事件。实际上，我们处理的是 ButtonBase.Click 事件，因为 ButtonBase 是定义 Click 事件的类，而 Button 控件继承了这个类。

关联事件的语法与关联属性的特性语法相同：

```
<Grid Name="contentGrid" ButtonBase.Click="contentGrid_Click"...>
  <Button Name="button1" ...>Button1</Button>
  <Button Name="button2" ...>Button2</Button>
  ...
  <Button Name="button1000" ...>Button1000</Button>
```

```
</Grid>
```

在事件处理程序中，从 `sender` 参数中获得 `Grid` 控件的一个引用，然后使用 `RoutedEventArgs.Source` 属性确定单击了哪个按钮，并做出相应的响应。这个事件仅在单击按钮时触发，而在单击 `Grid` 控件的背景时不会触发，因为 `Grid` 控件没有 `Click` 事件。

### 25.3.5 控件的布局

本章前面一直使用 `Grid` 元素布置控件，这主要是因为在创建新的 WPF 应用程序时，这是默认提供的控件。但是我们还没有充分利用这个类的全部功能，也没有学习能得到其他布局效果的其他布局容器。本节将详细介绍控件布局，因为这是 WPF 中需要掌握的一个基本概念。

所有的内容布局控件都派生于抽象类 `Panel`。这个类仅定义了一个容器，该容器可以包含派生于 `UIElement` 的对象集合。所有的 WPF 控件都派生于 `UIElement`。不能直接使用 `Panel` 类控制布局，但可以派生于它。另外，还可以使用下面派生于 `Panel` 的布局控件：

- **Canvas**：这个控件可以按任意方式定位子控件。它对子控件的定位没有任何限制，但也没有给子控件的定位提供任何帮助。
- **DockPanel**：这个控件可以把子控件停靠在它的 4 条边上。最后一个子控件会占用剩余的空间。
- **Grid**：前面已经介绍了这个控件如何灵活地定位子控件。但还没有介绍如何把这个控件的布局分成行和列，使控件在栅格布局中对齐。
- **StackPanel**：这个控件以水平或垂直布局来布置其子控件。
- **WrapPanel**：这个控件与 `StackPanel` 一样，也以水平或垂直布局来布置其子控件，但它不是仅在单行或单列上布置控件，而是根据可用的空间，允许把子控件放在多行或多列上。

下面详细介绍这些控件的用法。但首先需要理解几个基本概念：

- 控件如何以堆栈顺序显示
- 如何使用对齐、页边距和填充来定位控件及其内容
- 如何使用 `Border` 控件

#### 1. 堆栈顺序

容器控件包含多个子控件时，它们会以指定的堆栈顺序显示。读者应在绘图软件包中熟悉这个概念。理解堆栈顺序的最佳方式是假定每个控件放在一个玻璃盘上，而容器控件包含一叠这样的玻璃盘。因此，如果通过这些透明的玻璃从上向下看，就可以看到容器的内容。包含在容器中的控件重叠放置，因此看到的内容取决于玻璃盘的顺序。如果某个控件在该叠玻璃盘偏上的位置，就可以在重叠的区域中看到它。而位于该叠玻璃盘偏下位置的控件就会被其上的控件部分或全部覆盖掉。

用鼠标单击窗口时，这也会影响单击测试。考虑到控件的重叠，目标控件应总是位于堆栈最上面的那个控件。控件的堆栈顺序由它们显示在容器的子控件列表中的顺序确定。容器中的第一个子控件放在堆栈最下面的一层，最后一个控件放在堆栈最上面的一层。在第一个和最后一个子控件中间的其他子控件放在中间层上。在可以在 WPF 中使用的一些布局控件中，控件的堆栈顺序还有其他含义，如后面所述。



## 2. 对齐、页边距、填充和尺寸

前面的示例说明了 `Margin`、`HorizontalAlignment` 和 `VerticalAlignment` 如何在 `Grid` 容器中定位控件。还说明了如何使用 `Height` 和 `Width` 指定尺寸。这些属性和尚未探讨的 `Padding` 属性对所有的(或大多数)布局控件都很有用,但它们的使用方式不同。不同布局控件也可以为这些属性设置默认值。后面的各小节会介绍一些示例,但在此之前先介绍基本概念。

两个对齐属性确定了控件的对齐方式,但我们并没有列出这些属性的所有值。例如,`HorizontalAlignment` 可以设置为 `Left`、`Right`、`Center` 或 `Stretch`。`Left` 和 `Right` 会把控件定位在容器的左或右边界上,`Center` 把控件定位在中心,`Stretch` 会改变控件宽度,使其边界延伸到容器的边界上。`VerticalAlignment` 与此类似,其值有 `Top`、`Bottom`、`Center` 或 `Stretch`。

`Margin` 和 `Padding` 分别指定在控件的边界周围和控件边界的内部预留多少空间。前面的示例使用 `Margin` 指定控件相对于 `Grid` 左上角等地点的位置。这是有效的,因为把 `HorizontalAlignment` 设置为 `Left`,`VerticalAlignment` 设置为 `Top`,控件就会位于容器的左上角,而 `Margin` 在控件的边界周围插入了一个空隙。`Padding` 的用法类似,但在控件的内容与其边界之间空出了一些地方。这对 `Border` 控件尤其有用,如下一节所述。`Margin` 和 `Padding` 都可以指定为 4 个部分值(其形式是 `leftAmount`、`topAmount`、`rightAmount`、`bottomAmount`)或者指定为一个值(`Thickness` 值)。

`Height` 和 `Width` 常由其他属性控制。例如,把 `HorizontalAlignment` 设置为 `Stretch`,控件的 `Width` 属性就随着其容器宽度的变化而变化。

## 3. Border 控件

`Border` 控件是一个简单实用的容器控件,它包含一个子控件,而不像稍后介绍的更复杂的控件那样包含多个控件。这个子控件会完全填满 `Border` 控件。这似乎没有什么用,但可以使用 `Margin` 和 `Padding` 属性把 `Border` 定位在其容器中,把 `Border` 的内容定位在 `Border` 的边界中。也可以设置 `Border` 的 `Background` 属性,使之可见。稍后介绍这个控件。

## 4. Canvas 控件

如前所述,`Canvas` 控件对控件的定位提供了完全的自由。另外,用于某个子元素的 `HorizontalAlignment` 和 `VerticalAlignment` 属性不会影响其他元素的定位。

使用 `Margin` 可以在 `Canvas` 中定位元素,如前面的示例所示。但更好的方式是使用 `Canvas` 类的 `Canvas.Left`、`Canvas.Top`、`Canvas.Right` 和 `Canvas.Bottom` 关联属性:

```
<Canvas ...>
  <Button Canvas.Top="10" Canvas.Left="10" ...>Button1</Button>
</Canvas>
```

上面的代码定位了一个按钮,使其顶边距离 `Canvas` 的顶边 10 个像素,其左边界距离 `Canvas` 的左边界 10 个像素。注意 `Top` 和 `Left` 属性的优先级高于 `Bottom` 和 `Right`。例如,如果指定了 `Top` 和 `Bottom` 属性,就忽略 `Bottom` 属性。

图 25-10 显示了 `Canvas` 控件中定位的两个 `Rectangle` 控件,但窗口有两个不同尺寸。

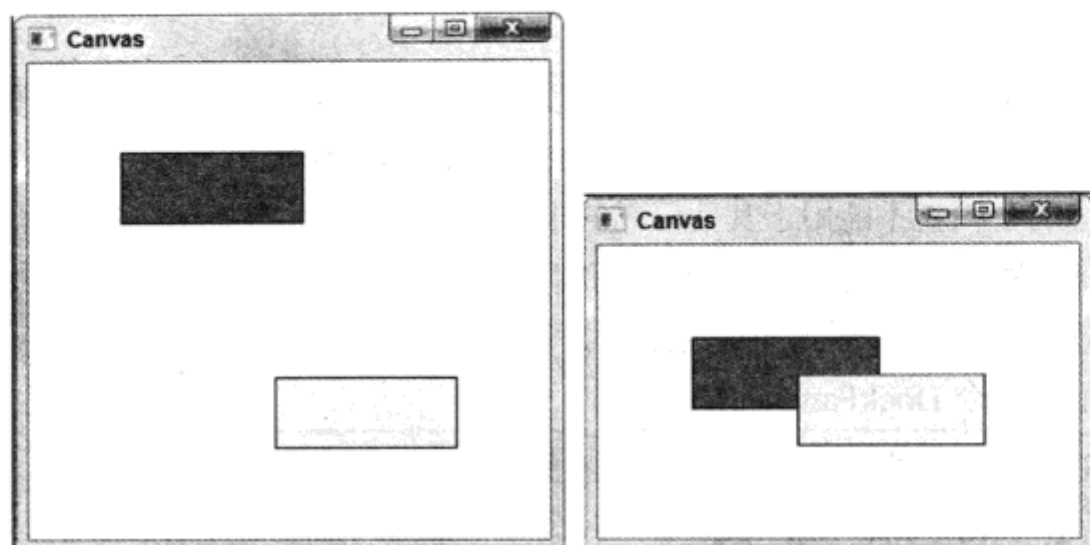


图 25-10



本节的所有示例布局都在本章下载代码的 LayoutExamples 项目中。

一个 Rectangle 相对于 Canvas 控件的左上角定位，另一个 Rectangle 相对于 Canvas 控件的右下角定位。在重新设置窗口的大小时，这些相对位置保持不变。还可以看出 Rectangle 控件的堆栈顺序的重要性。右下角的 Rectangle 在堆栈顺序中比较高，所以两个 Rectangle 重叠时，就可以看到右下角的 Rectangle。

这个示例的代码如下：



可从  
wrox.com  
下载源代码

```
<Canvas Background="AliceBlue">
  <Rectangle Canvas.Left="50" Canvas.Top="50" Height="40" Width="100"
    Stroke="Black" Fill="Chocolate" />
  <Rectangle Canvas.Right="50" Canvas.Bottom="50" Height="40" Width="100"
    Stroke="Black" Fill="Bisque" />
</Canvas>
```

代码段 LayoutExamples\CanvasWindow.xaml

## 5. DockPanel 控件

顾名思义，DockPanel 控件允许把控件停靠在它的一个边界上。读者即使以前从来没有注意过，也应很熟悉这类布局。Word 中的 Ribbon 控件就总是位于 Word 窗口的顶部，VS 和 VCE 中的各个窗口也是用这种方式定位的。在 VS 和 VCE 中，拖动窗口就会改变窗口的停靠方式。

DockPanel 有一个关联属性 DockPanel.Dock，子控件可以使用该属性指定它停靠在哪个边上。这个属性可以设置为 Left、Top、Right 和 Bottom。

DockPanel 中控件的堆栈顺序非常重要，因为每次把控件停靠在一条边上时，都会相应地减小后续子控件的可用空间。例如，把一个工具条停靠在 DockPanel 的顶部，再把第二个工具条停靠在 DockPanel 的左边。第一个控件会延伸到 DockPanel 显示区域的整个顶部，但第二个控件只能从第一个工具条的底部开始沿着 DockPanel 的左边界延伸到 DockPanel 的底部。

在前面的子控件都定位好后，最后一个指定的子控件通常会占据剩余的空间(可以控制这个



行为)。

在 DockPanel 中定位控件时, 该控件占用的空间可能比 DockPanel 给该控件预留的区域小。例如, 如果把 Width 为 100、Height 为 50、HorizontalAlingment 为 Left 的按钮停靠在 DockPanel 的顶部, 则该按钮右边的空间就不能用于其他停靠的子控件。另外, 如果按钮控件的 Margin 为 20, 就要在 DockPanel 顶部预留总共 90 个像素(控件的高度加上边距和下边距)。在用 DockPanel 进行布局时, 要考虑到这种情况; 否则就得不到期望的结果。

图 25-11 显示了一个 DockPanel 布局示例。

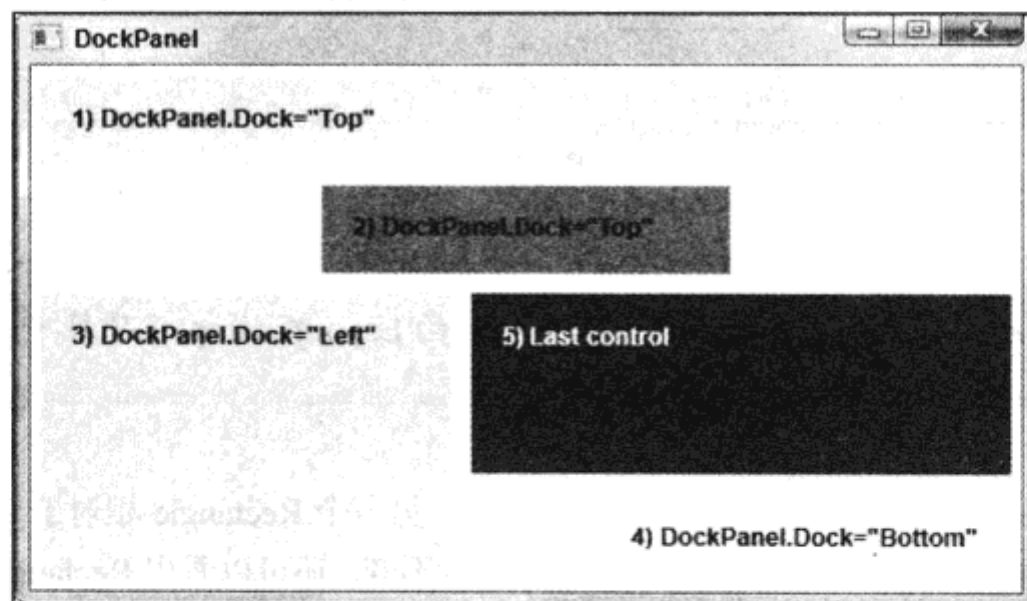


图 25-11

这个布局的代码如下所示:



可从  
wrox.com  
下载源代码

```
<DockPanel Background="AliceBlue">
  <Border DockPanel.Dock="Top" Padding="10" Margin="5"
    Background="Aquamarine" Height="45">
    <Label>1) DockPanel.Dock="Top"</Label>
  </Border>
  <Border DockPanel.Dock="Top" Padding="10" Margin="5"
    Background="PaleVioletRed" Height="45" Width="200">
    <Label>2) DockPanel.Dock="Top"</Label>
  </Border>
  <Border DockPanel.Dock="Left" Padding="10" Margin="5"
    Background="Bisque" Width="200">
    <Label>3) DockPanel.Dock="Left"</Label>
  </Border>
  <Border DockPanel.Dock="Bottom" Padding="10" Margin="5"
    Background="Ivory" Width="200" HorizontalAlignment="Right">
    <Label>4) DockPanel.Dock="Bottom"</Label>
  </Border>
  <Border Padding="10" Margin="5" Background="BlueViolet">
    <Label Foreground="White">5) Last control</Label>
  </Border>
</DockPanel>
```

代码段 LayoutExamples\DockPanelWindow.xaml

这段代码在示例布局中使用前面介绍的 Border 控件清楚地划分了停靠控件的区域, 再使用 Label 控件输出简单的信息文本。为了理解该布局, 必须自上而下读取代码, 依次查看每个控件:

(1) 第一个 Border 控件停靠在 DockPanel 的顶部。DockPanel 中被占据的总区域是顶部的 55 个像素( $\text{Height} + 2 \times \text{Margin}$ )。注意, Padding 属性对这个布局没有影响, 因为它在 Border 边界的内部, 但这个属性控制着其内嵌 Label 控件的定位。如果没有 Height 或 Width 属性的限制, Border 控件就会在 DockPanel 中延伸, 填满 DockPanel 顶部的所有可用空间。

(2) 第二个 Border 控件也停靠在 DockPanel 的顶部, 占据了显示区域顶部的另外 55 个像素。这个 Border 控件还包含 Width 属性, 使 Border 仅占据 DockPanel 的部分宽度。它位于中心, 因为 DockPanel 中的 HorizontalAlignment 的默认值是 Center。

(3) 第三个 Border 控件停靠在 DockPanel 的左边, 占据显示区域左边的 210 个像素。

(4) 第四个 Border 控件停靠在 DockPanel 的底部, 占据 30 个像素加上它包含的 Label 控件的高度。这个高度由 Margin、Padding 和 Border 控件的内容确定, 因为它没有明确指定。Border 控件锁定在 DockPanel 的右下角, 因为它的 HorizontalAlignment 为 Right。

(5) 第五个也是最后一个 Border 控件填满了剩余的空间。

运行这个示例, 试着改变其内容的尺寸。注意, 控件的堆栈顺序越大, 获得其空间的优先级就越高。缩小窗口, 第五个 Border 控件会很快被堆栈顺序较大的控件覆盖, 使用 DockPanel 控制布局时要小心避免这种情况, 例如, 可以设置窗口的最小尺寸。

## 6. Grid 控件

Grid 控件可以包含多行和多列, 用于布置子控件。本章前面已经使用过几次 Grid 控件, 但所有示例都使用单行单列的 Grid 控件。要添加更多行和列, 必须使用 RowDefinitions 和 ColumnDefinitions 属性, 它们分别是 RowDefinition 和 ColumnDefinition 对象的集合, 用属性元素语法指定:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  ...
</Grid>
```

这段代码定义了一个包含 3 行 2 列的 Grid 控件。注意, 这里不需要额外的信息。在这段代码中, Grid 控件重新设置大小时, 都会自动重置每一行和每一列的大小。每一行的高度都是 Grid 控件高度的 1/3, 每一列的宽度都是 Grid 控件宽度的一半。把 Grid.ShowGridlines 属性设置为 true, 就可以在 Grid 的单元格之间显示线条。

Width、Height、MinWidth、MaxWidth、MinHeight 和 MaxHeight 可用于控制尺寸的重置。例如, 设置列的 Width 属性可以确保该列的宽度保持不变。也可以把列的 Width 属性设置为\*, 表示“计算完其他列的宽度后填满剩余的空间”。这实际上是默认值。多个列的宽度都是\*时, 剩余的空间就由它们平分。\*值也可以用于行的 Height 属性。Height 和 Width 的另一个值是 Auto, 表示根据控件的内容设置行或列的大小。也可以使用 GridSplitter 控件, 让用户通过单击和拖动操作来定制行和列的尺寸。

Grid 控件的子控件可以使用关联属性 Grid.Column 和 Grid.Row 指定它们包含在什么单元格中。这些属性的默认值都是 0，所以如果忽略这两个属性，子控件就位于左上单元格中。子控件也可以使用 Grid.ColumnSpan 和 Grid.RowSpan 放在表的多个单元格中，此时左上单元格由 Grid.Column 和 Grid.Row 指定。

图 25-12 中的 Grid 控件包含多个椭圆和一个 GridSplitter，且窗口有两个不同尺寸。

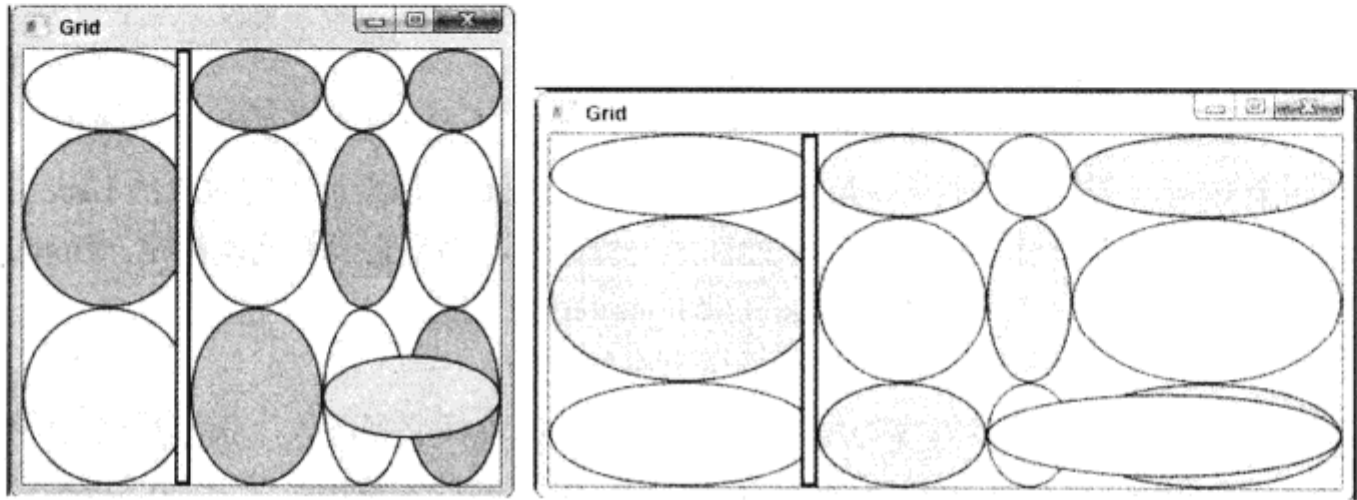


图 25-12

所用的代码如下：



```
<Grid Background="AliceBlue">
  <Grid.ColumnDefinitions>
    <ColumnDefinition MinWidth="100" MaxWidth="200" />
    <ColumnDefinition MaxWidth="100" />
    <ColumnDefinition Width="50" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition MinHeight="100" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Ellipse Grid.Row="0" Grid.Column="0" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="0" Grid.Column="1" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="0" Grid.Column="2" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="0" Grid.Column="3" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="0" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="1" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="2" Fill="BurlyWood"
    Stroke="Black" />
  <Ellipse Grid.Row="1" Grid.Column="3" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="0" Fill="BlanchedAlmond"
    Stroke="Black" />
  <Ellipse Grid.Row="2" Grid.Column="1" Fill="BurlyWood"
    Stroke="Black" />
```

```

        Stroke="Black" />
<Ellipse Grid.Row="2" Grid.Column="2" Fill="BlanchedAlmond"
        Stroke="Black" />
<Ellipse Grid.Row="2" Grid.Column="3" Fill="BurlyWood"
        Stroke="Black" />
<Ellipse Grid.Row="2" Grid.Column="2" Grid.ColumnSpan="2" Fill="Gold"
        Stroke="Black" Height="50"/>
<GridSplitter Grid.RowSpan="3" Width="10" BorderThickness="2">
    <GridSplitter.BorderBrush>
        <SolidColorBrush Color="Black" />
    </GridSplitter.BorderBrush>
</GridSplitter>
</Grid>

```

代码段 LayoutExamples\GridWindow.xaml

这段代码使用行和列定义上的各种属性，在重置显示区域的大小时，获得了有趣的效果，所以最好测试一下。

首先考虑行。顶行的高度是固定的 50 像素，第二行的最小高度设置为 100 像素，第三行填满了剩余的空间。这表示如果 Grid 的高度小于 150 像素，第三行就是不可见的。Grid 的高度在 150~250 像素之间时，只有第三行的尺寸会在 0~100 像素之间变化。这是因为剩余的空间应计算为总高度减去高度固定的行的高度之和。剩余的空间位于第二行和第三行之间，但因为第二行的最小高度是 100 像素，所以它的高度不会变化，除非 Grid 的总高度达到 250 像素。最后，Grid 的总高度超过 250 像素时，第二行和第三行会分享剩余的空间，所以它们的高度都等于或大于 100 像素。

接着分析列。只有第三列的尺寸是固定的 50 像素。第一列和第二列的总宽度是 300 像素。因此，在 Grid 控件的总宽度超过 550 像素时，只有第四列的尺寸会增大。为了弄明白这一点，应考虑列可用的像素值是多大，它们是如何分配的。首先，把 50 个像素分配给第三列，剩余的 500 像素分配给其余的列。第三列的最大宽度是 100 像素，第一列和第四列的宽度就剩下 400 像素。第一列的最大宽度是 200 像素，所以即使宽度超过了这个值，第一列也不会占用更多的空间。而第四列的尺寸会增加。

注意这个示例的另外两点。首先，最后一个定义的椭圆占据了第三列和第四列，以演示 Grid.ColumnSpan 的用法。其次，提供了一个 GridSplitter，允许重置第一列和第二列的大小。但是，一旦 Grid 控件的总宽度超过 550 像素，这个 GridSplitter 就不能设置这些列的大小，因为第一列和第二列都不能增大尺寸了。

GridSplitter 控件很有用，但其外观很呆板。这是一个需要设置样式的控件，至少要把其 Background 属性设置为 Transparent，使之不可见。

如果窗口中有多个 Grid 控件，还可以在行或列定义上使用 ShareSizeGroup 属性为行或列定义共享的尺寸组。ShareSizeGroup 属性可以设置为字符串标识符。例如，如果在一个 Grid 控件中，共享尺寸组中的一列改变了，则该组中另一个 Grid 控件的列也会改变，以匹配前一个 Grid 控件。通过 Grid.IsShareSizeScope 属性可以启用或禁用这个功能。

## 7. StackPanel 控件

了解到 Grid 控件的复杂性之后，读者会发现 StackPanel 控件是一个较简单的布局控件。可以将 StackPanel 控件看作 DockPanel 的一个删节版本，因为子控件停靠的边是固定的。这两个控件的另

一个区别是 StackPanel 的最后一个子控件不填满剩余的空间。但是，控件会默认延伸到 StackPanel 控件的边界上。

控件堆叠的方向由 3 个属性确定。Orientation 可以设置为 Horizontal 或 Vertical，HorizontalAlignment 和 VerticalAlignment 可以确定控件是沿着 StackPanel 的顶边、底边、左边还是右边定位。给所使用的对齐属性设置 Center 值，甚至可以使控件在 StackPanel 的中心处堆叠。

图 25-13 显示了两个 StackPanel 控件，它们都包含 3 个按钮。StackPanel 控件使用包含两行一列的 Grid 控件定位。

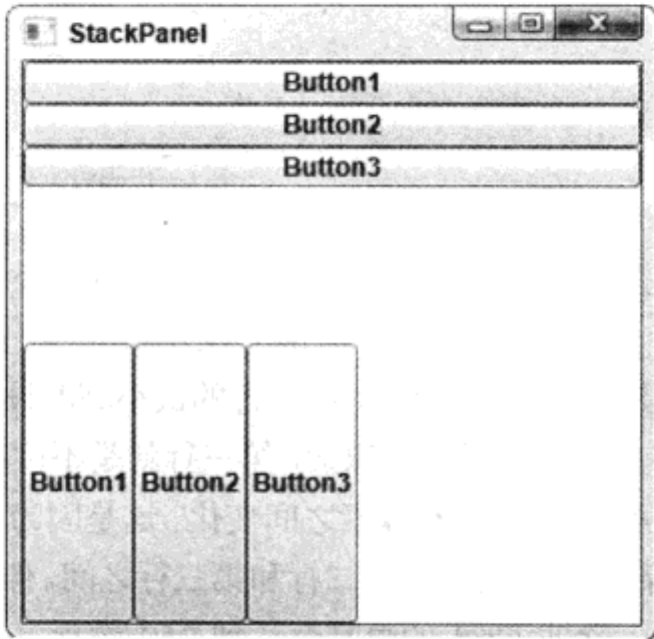


图 25-13

这里使用的代码如下：



可从  
wrox.com  
下载源代码

```
<Grid Background="AliceBlue">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <StackPanel Grid.Row="0">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
  <StackPanel Grid.Row="1" Orientation="Horizontal">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</Grid>
```

代码段 LayoutExamples\StackPanelWindow.xaml

使用 StackPanel 来布局时，经常需要添加滚动条，以便查看包含在 StackPanel 中的所有控件。WPF 为我们做了许多工作。使用 ScrollViewer 控件就可以添加滚动条——只需在这个控件中包含 StackPanel:

```
<Grid Background="AliceBlue">
  <Grid.RowDefinitions>
```

```

<RowDefinition />
<RowDefinition />
</Grid.RowDefinitions>
<ScrollView>
  <StackPanel Grid.Row="0">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</ScrollView>
<StackPanel Grid.Row="1" Orientation="Horizontal">
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
</StackPanel>
</Grid>

```

可以使用更复杂的技术以不同的方式滚动，或者用编程方式滚动，但上述代码常常就是需要编写的代码。

## 8. WrapPanel 控件

WrapPanel 实际上是 StackPanel 的一个扩展版本，其中“不合适”的控件会移动到附加的行或列上。图 25-14 显示的 WrapPanel 控件包含多个形状，而窗口设置为两个不同的尺寸。

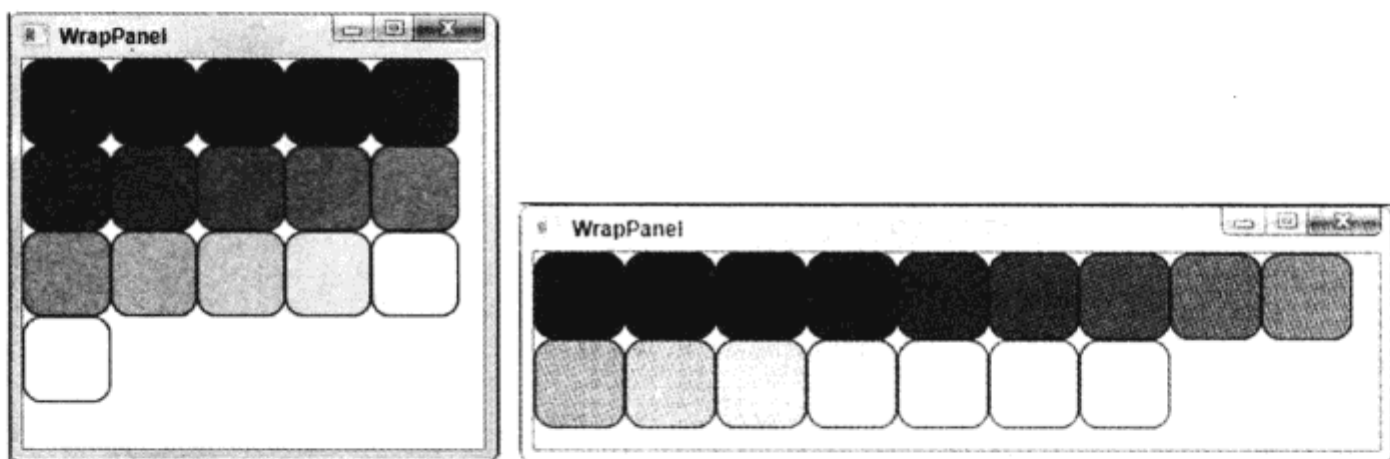


图 25-14

代码的缩减版本如下：



可从  
wrox.com  
下载源代码

```

<WrapPanel Background="AliceBlue">
  <Rectangle Fill="#FF000000" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  <Rectangle Fill="#FF111111" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  <Rectangle Fill="#FF222222" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  ...
  <Rectangle Fill="#FFFFFFF" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
</WrapPanel>

```

代码段 LayoutExamples\WrapPanelWindow.xaml



WrapPanel 控件是创建动态布局的一种好方法，允许用户精确控制内容的显示方式。

### 25.3.6 控件的样式

WPF 的一个杰出特性是它允许设计人员全面控制用户界面的外观和操作方式。其核心是可以给控件设置任意二维或三维样式。前面一直使用的是.NET 3.5 为控件提供的基本样式，但控件可以使用的样式是无穷多的。

本节介绍两种基本技术：

- **样式：**成批应用于控件的属性组
- **模板：**用于建立控件外观的控件

这两种技术有些重叠，因为样式可以包含模板。

#### 1. 样式

WPF 控件有一个 Style 属性(继承自 FrameworkElement)，它可以设置为 Style 类的实例。Style 类相当复杂，具有高级样式化功能，但其核心是一组 Setter 对象。每个 Setter 对象都根据 Property 属性(要设置的属性名)及 Value 属性(属性要设置的值)设置属性值。可以把在 Property 中使用的名称完全限定为控件类型(如 Button.Foreground)，也可以设置 Style 对象的 TargetType 属性(如 Button)，使之可以解析属性名。

下面的代码说明了如何使用 Style 对象设置 Button 控件的 Foreground 属性：

```
<Button>
  Click me!
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Foreground">
        <Setter.Value>
          <SolidColorBrush Color="Purple" />
        </Setter.Value>
      </Setter>
    </Style>
  </Button.Style>
</Button>
```

显然，在这段代码中，以通常的方式设置按钮的 Foreground 属性要简单得多。把样式转换为资源后，样式会非常有用，因为资源是可以重用的。本章后面将介绍其转换过程。

#### 2. 模板

控件用模板构建，而模板是可以定制的。模板包含一个控件层次结构，用于建立控件的显示外观，该外观可能包含控件的内容显示器，例如，显示内容的按钮。

控件的模板常常存储在其 Template 属性中，Template 属性是 ControlTemplate 类的一个实例。ControlTemplate 类包含 TargetType 属性，它可以设置为用于定义模板的控件类型，这个属性可以包含单个控件，也可以是一个容器，例如 Grid，所以它不会限制我们的操作。

一般使用样式来设置类的模板。这只需如下方式为要使用的控件提供 Template 属性：

```
<Button>
```



```

Click me!
<Button.Style>
  <Style TargetType="Button">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          ...
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Button.Style>
</Button>

```

一些控件需要多个模板。例如, CheckBox 控件使用一个模板表示其复选框(CheckBox.Template), 使用另一个模板输出复选框旁边的文本(CheckBox.ContentTemplate)。

需要内容显示器的模板可以在输出内容的位置上包含 ContentPresenter 控件。一些控件, 尤其是输出数据项集合的控件, 使用另一种技术, 本章不讨论该技术。

在与资源合并使用时, 替代模板是很有用的。但是, 设置控件的样式是一种非常常见的技术, 需要用示例来说明。

### 试一试: 使用样式和模板

- (1) 创建一个新 WPF 应用程序 Ch25Ex03, 将其保存在 C:\BegVCSharp\Chapter25 目录中。
- (2) 修改 MainWindow.xaml 的代码, 如下所示:



可从  
wrox.com  
下载源代码

```

<Window x:Class="Ch25Ex03.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Nasty Button" Height="150" Width="550">
  <Grid Background="Black">
    <Button Margin="20" Click="Button_Click">
      Would anyone use a button like this?
    <Button.Style>
      <Style TargetType="Button">
        <Setter Property="FontSize" Value="18" />
        <Setter Property="FontFamily" Value="arial" />
        <Setter Property="FontWeight" Value="bold" />
        <Setter Property="Foreground">
          <Setter.Value>
            <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
              <LinearGradientBrush.GradientStops >
                <GradientStop Offset="0.0" Color="Purple" />
                <GradientStop Offset="0.5" Color="Azure" />
                <GradientStop Offset="1.0" Color="Purple" />
              </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
          </Setter.Value>
        </Setter>
        <Setter Property="Template">
          <Setter.Value>
            <ControlTemplate TargetType="Button">

```

```

<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="50" />
    <ColumnDefinition />
    <ColumnDefinition Width="50" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition MinHeight="50" />
  </Grid.RowDefinitions>
  <Ellipse Grid.Column="0" Height="50">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
          <GradientStop Offset="0.0" Color="Yellow" />
          <GradientStop Offset="1.0" Color="Red" />
        </RadialGradientBrush.GradientStops>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Grid Grid.Column="1">
    <Rectangle RadiusX="10" RadiusY="10">
      <Rectangle.Fill>
        <RadialGradientBrush>
          <RadialGradientBrush.GradientStops>
            <GradientStop Offset="0.0" Color="Yellow" />
            <GradientStop Offset="1.0" Color="Red" />
          </RadialGradientBrush.GradientStops>
        </RadialGradientBrush>
      </Rectangle.Fill>
    </Rectangle>
    <ContentPresenter Margin="20,0,20,0"
      HorizontalAlignment="Center"
      VerticalAlignment="Center" />
  </Grid>
  <Ellipse Grid.Column="2" Height="50">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
          <GradientStop Offset="0.0" Color="Yellow" />
          <GradientStop Offset="1.0" Color="Red" />
        </RadialGradientBrush.GradientStops>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Button.Style>
</Button>
</Grid>
</Window>

```

代码段 Ch25Ex03\MainWindow.xaml

(3) 修改 MainWindow.xaml.cs 中的代码，如下所示：



```
public partial class MainWindow : Window
{
    ...

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Button clicked.");
    }
}
```

代码段 Ch25Ex03\MainWindow.xaml.cs

(4) 运行应用程序，单击一次按钮，结果如图 25-15 所示。

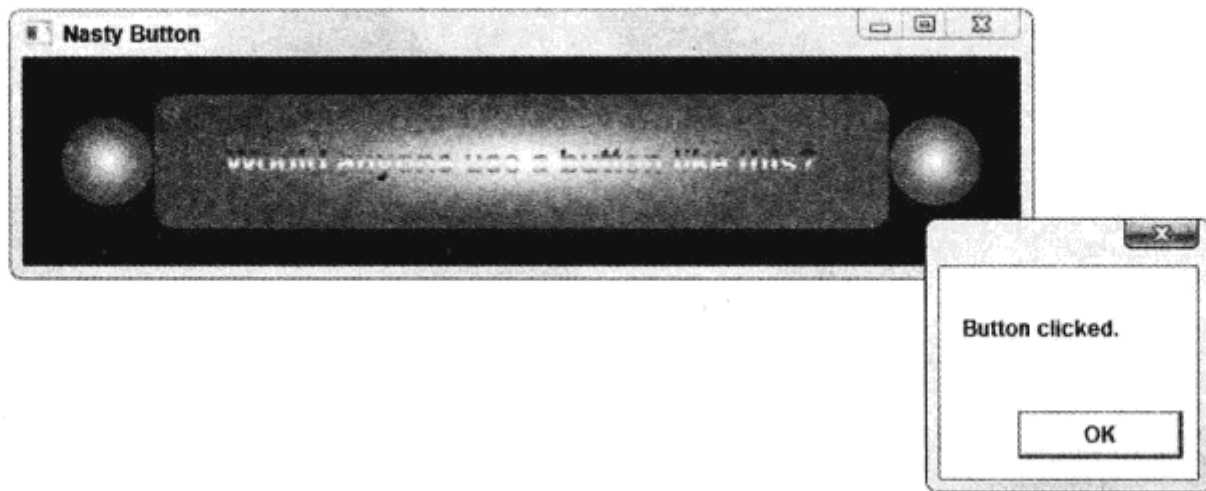


图 25-15

#### 示例的说明

这个示例中的按钮非常难看。但是，抛去外观方面的因素，这个示例其实说明了可以在 WPF 中毫不费力地全面控制按钮的外观。但改变按钮的模板时，注意按钮的功能没有变化。即可以单击按钮，在事件处理程序中响应该单击。

注意在这里使用的模板上没有实现与 Windows 按钮关联的一些操作。例如，把鼠标停放在按钮上或单击按钮时，没有可视化的反馈。这个按钮无论是否获得焦点，其外观都是相同的。为了得到这些效果，需要学习触发器，这是下一节的内容。

但在此之前，先详细分析一下示例代码，主要关注样式和模板，看看模板是如何创建的。

刚开始的代码很普通，用于显示按钮控件：

```
<Button Margin="20" Click="Button_Click">
    Would anyone use a button like this?
```

这给按钮提供了基本属性和内容。接着，Style 属性设置为 Style 对象，开始为按钮控件设置 3 个简单的字体属性：

```
<Button.Style>
    <Style TargetType="Button">
        <Setter Property="FontSize" Value="18" />
        <Setter Property="FontFamily" Value="arial" />
        <Setter Property="FontWeight" Value="bold" />
```

接着，使用属性元素语法设置 `Button.Foreground` 属性，因为使用了一个笔刷：

```
<Setter Property="Foreground">
  <Setter.Value>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.0" Color="Purple" />
        <GradientStop Offset="0.5" Color="Azure" />
        <GradientStop Offset="1.0" Color="Purple" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Setter.Value>
</Setter>
```

Style 对象的剩余代码将 `Button.Template` 属性设置为 `ControlTemplate` 对象：

```
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          ...
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Button.Style>
</Button>
```

模板代码可以汇总为一个 `Grid` 控件，该控件在一行中有 3 个单元格。这些单元格包含一个 `Ellipse`、一个 `Rectangle`、模板的 `ContentPresenter` 和另一个 `Ellipse`：

```
<Grid>
  <Ellipse Grid.Column="0" Height="50">
    ...
  </Ellipse>
  <Grid Grid.Column="1">
    <Rectangle RadiusX="10" RadiusY="10">
      ...
    </Rectangle>
    <ContentPresenter Margin="20,0,20,0"
      HorizontalAlignment="Center"
      VerticalAlignment="Center" />
  </Grid>
  <Ellipse Grid.Column="2" Height="50">
    ...
  </Ellipse>
</Grid>
```

这些代码都较为简单，下面要进一步分析它们。

### 25.3.7 触发器

在本章的第一个示例中，学习了触发器可以把事件关联到动作上。WPF 中的事件可以包含所有的操作，包括按钮单击、应用程序启动和停止等。在 WPF 中有几类触发器，它们都继承自基类 `TriggerBase`。示例中使用的触发器类型是 `EventTrigger`。`EventTrigger` 类包含一个动作集合，每个动

作都是派生于基类 `TriggerAction` 的一个对象。这些动作都在激活触发器时执行。

在 WPF 中，并不是许多类都继承自 `TriggerAction`，当然，我们可以定义自己的类。可以使用 `EventTrigger` 通过 `BeginStoryboard` 动作触发动画，通过 `ControllableStoryboardAction` 处理故事板，通过 `SoundPlayerAction` 触发声音效果。最后一个触发器在动画中最常用，所以下一节介绍它。

每个控件都有 `Triggers` 属性，用于直接在该控件上定义触发器。也可以在该层次结构中进一步定义触发器，例如，在前面所示的 `Window` 对象上定义触发器。在给控件设置样式时，最常用的触发器类型是 `Trigger` (也可以使用 `EventTrigger` 引发控件动画)。`Trigger` 类用于设置属性，以响应其他属性的变化，在 `Style` 对象中使用该类时特别有用。

`Trigger` 对象的配置如下：

- 要定义 `Trigger` 对象监控的属性，使用 `Trigger.Property` 属性。
- 要定义 `Trigger` 对象的激活时间，设置 `Trigger.Value` 属性。
- 要定义 `Trigger` 对象执行的动作，把 `Trigger.Setters` 属性设置为 `Setters` 对象的集合。

这里的 `Setter` 对象就是前面“样式”一节中的 `Setter` 对象。

例如，下面的触发器将检查属性 `MyBooleanValue` 的值，在该属性为 `true` 时，把 `Opacity` 属性的值设置为 0.5：

```
<Trigger Property="MyBooleanValue" Value="true">
  <Setter Property="Opacity" Value="0.5" />
</Trigger>
```

这段代码并没有给出许多信息，因为它未与任何控件或样式关联。下面的代码较易理解，因为它说明了 `Trigger` 在 `Style` 对象中的用法：

```
<Style TargetType="Button">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="true">
      <Setter Property="Foreground" Value="Yellow" />
    </Trigger>
  </Style.Triggers>
</Style>
```

这段代码在 `Button.IsMouseOver` 属性为 `true` 时，把按钮控件的 `Foreground` 属性改为 `Yellow`。`IsMouseOver` 是几个非常有用的属性之一，可以用作查找控件信息和控件状态的快捷方式。顾名思义，如果鼠标停放在控件上，这个属性就是 `true`。此时可以给鼠标的停放操作编码。类似于这个属性的其他属性有 `IsFocused`，它确定控件是否获得焦点；`IsHitTestVisible`，它指定是否能单击某个控件(即该控件没有被堆栈顺序更高的控件遮住)；以及 `IsPressed`，它指定按钮是否被按下。最后一个属性仅应用于继承自 `ButtonBase` 的按钮，而其他属性可以应用于所有控件。

除了 `Style.Triggers` 属性之外，还可以使用 `ControlTemplate.Triggers` 属性获得许多信息。这个属性可以为包含触发器的控件创建模板。所以默认的 `Button` 模板可以响应鼠标的停放、单击和焦点的变化。我们必须修改这个模板，自己实现该功能。

### 25.3.8 动画

动画是用故事板创建的。毫无疑问，定义动画的最佳方式是使用 `Expression Blend` 等设计器。也可以直接编辑 XAML 代码，根据代码隐藏中的含义来定义动画(因为 XAML 只是建立 WPF 对象

模型的一种方式)。

用 Storyboard 对象定义故事板，Storyboard 对象包含一个或多个时间线。定义时间线可以使用关键帧，也可以使用几个更简单的对象来封装整个动画。复杂的故事板甚至可以包含嵌套的故事板。

如示例所示，Storyboard 包含在一个资源目录中，所以必须用 x:Key 属性标识它。

在故事板的时间线中，可以连续改变应用程序中的任意元素的动画属性，属性类型可以是 double、Point 或 Color。这涵盖了需要连续改变的所有元素属性，所以非常灵活。有一些操作不能执行，例如，用一种笔刷完全替代另一种笔刷，但只要有了这 3 种类型，就总有办法达到需要的任何效果。

这 3 种类型都有两个关联的时间线控件，可用作 Storyboard 的子控件。这 6 个控件是 DoubleAnimation、DoubleAnimationUsingKeyFrames、PointAnimation、PointAnimationUsingKeyFrames、ColorAnimation 和 ColorAnimationUsingKeyFrames。每个时间线控件都可以使用关联属性 Storyboard.TargetName 和 Storyboard.TargetProperty 关联到特定控件的特定属性上。例如，如果要连续改变一个 Rectangle 控件(其 Name 属性是 MyRectangle)的 Width 属性，就可以把 Storyboard.TargetName 和 Storyboard.TargetProperty 属性设置为 MyRectangle 和 Width。也可以使用 DoubleAnimation 或 DoubleAnimationUsingKeyFrames 连续改变这个属性。

Storyboard.TargetProperty 属性可以解释相当高级的语法，以便定位动画中感兴趣的属性。在本章开头的示例中，为两个关联属性使用了下面的值：

```
Storyboard.TargetName="ellipse1"
Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0]
.(RotateTransform.Angle)"
```

控件 ellipse1 的类型是 Ellipse，TargetProperty 指定椭圆在变换时旋转的角度。这个角度通过 Ellipse 的 RenderTransform 属性(继承自 UIElement)和 TransformGroup 对象的第一个子对象(就是这个属性的值)来定位。第一个子对象是 RotateTransform，角度是这个对象的 Angle 属性。

这个语法比较长，但用起来很简单。最困难的地方是确定给定的属性继承自哪个基类，但对象浏览器可以提供这方面的帮助。

下面看看无关键帧的简单动画时间线，然后再介绍使用关键帧的时间线。

1. 没有关键帧的时间线

没有关键帧的时间线是 DoubleAnimation、PointAnimation 和 ColorAnimation。这些时间线有相同的属性名，但这些属性的类型随时间线的类型而异(注意，所有的持续时间属性在 XAML 代码中都以格式[days.]hours:minutes:seconds 指定)，如表 25-2 所示。

表 25-2

属 性	用 法
Name	时间线的名称，用于在其他地方引用它
BeginTime	在触发故事板之后、时间线启动之前的时间
Duration	时间线的持续时间
AutoReverse	时间线完成时是否倒回，是否把属性返回为初始值。这个属性是一个布尔值



(续表)

属 性	用 法
RepeatBehavior	把这个属性设置为指定的持续时间，会使时间线根据指定的值重复执行——一个整数后跟 x(例如 5x)表示时间线重复指定的次数；也可以使用 Forever，让时间线重复执行到暂停或停止故事板为止
FillBehavior	如果时间线完成时，故事板仍旧在执行，时间线该如何操作。使用 HoldEnd 可以在时间线完成时不改变属性值(默认)，使用 Stop 可以把属性值返回为初始值
SpeedRatio	控制动画相对于其他属性的指定值的执行速度。默认为 1，但可以在其他代码中修改它，加快或减慢动画的执行速度
From	在动画开始时属性的初始值。可以忽略这个值，使用属性的当前值
To	在动画结束时属性的最终值。可以忽略这个值，使用属性的当前值
By	使用这个值使属性从当前值连续变化为当前值与指定值之和。这个属性可以单独使用，也可以与 From 属性结合使用

例如，下面的时间线使 Rectangle 控件(其 Name 属性是 MyRectangle)的 Width 属性在 5 秒内在 100~200 之间连续变化：

```
<Storyboard x:Key="RectangleExpander">
  <DoubleAnimation Storyboard.TargetName="MyRectangle"
    Storyboard.TargetProperty="Width" Duration="00:00:05"
    From="100" To="200" />
</Storyboard>
```

2. 有关键帧的时间线

有关键帧的时间线是 DoubleAnimationUsingKeyFrames、PointAnimationUsingKeyFrames 和 ColorAnimationUsingKeyFrames。这些时间线类使用的属性与上一节的时间线类的相同，但没有 From、To 和 By 属性，而是有一个 KeyFrames 属性(它是关键帧对象的集合)。

这些时间线可以包含任意多个关键帧，每个关键帧都可以用不同的方式改变属性值。每类时间线都有 3 种关键帧：

- **Discrete**：离散的关键帧会使连续变化的值跳到指定的值上，没有任何切换。
- **Linear**：线性的关键帧会使连续变化的值以线性变换方式连续改变为指定的值。
- **Spline**：样条曲线的关键帧会使连续变化的值按照三次贝塞尔曲线函数定义的非线性变换方式，连续改变为指定的值。

因此关键帧对象有 9 种：DiscreteDoubleKeyFrame、LinearDoubleKeyFrame、SplineDoubleKeyFrame、DiscreteColorKeyFrame、LinearColorKeyFrame、SplineColorKeyFrame、DiscretePointKeyFrame、LinearPointKeyFrame 和 SplinePointKeyFrame。

关键帧类的 3 个属性与上一节介绍的时间线类的相同，但样条曲线的关键帧还有一个额外的属性，如表 25-3 中所示。



表 25-3

属 性	用 法
Name	关键帧的名称, 用于在其他位置引用关键帧
KeyTime	关键帧的位置, 表示为时间线启动后经历的时间
Value	到达关键帧时的属性值, 或到达关键帧时应设置的属性值
KeySpline	两组 cplx,cply cp2x,cp2y 形式的数字, 它们定义了用于连续改变属性的三次贝塞尔函数(仅用于样条曲线的关键帧)

例如, 连续改变一个方块中 `Ellipse` 的 `Center` 属性(其类型是 `Point`), 就可以建立该 `Ellipse` 的位置动画:

```
<Storyboard x:Key="EllipseMover">
  <PointAnimationUsingKeyFrames Storyboard.TargetName="MyEllipse"
    Storyboard.TargetProperty="Center" RepeatBehavior="Forever">
    <LinearPointKeyFrame KeyTime="00:00:00" Value="50,50" />
    <LinearPointKeyFrame KeyTime="00:00:01" Value="100,50" />
    <LinearPointKeyFrame KeyTime="00:00:02" Value="100,100" />
    <LinearPointKeyFrame KeyTime="00:00:03" Value="50,100" />
    <LinearPointKeyFrame KeyTime="00:00:04" Value="50,50" />
  </PointAnimationUsingKeyFrames>
</Storyboard>
```

`Point` 值在 XAML 代码中以 `x,y` 的形式指定。

25.3.9 静态和动态资源

WPF 的另一个优秀特性是可以定义资源, 如控件样式和模板, 它们可以在应用程序中重用。如果在正确的地方定义资源, 还可以在多个应用程序中使用它们。

资源定义为 `ResourceDictionary` 对象中的项。顾名思义, 这是指定了键的对象集合。因此在本章前面的示例代码中定义资源时使用 `x:Key` 特性: 指定与资源关联的键。可以在多个不同位置访问 `ResourceDictionary` 对象。可以把资源包含在本地控件中、本地窗口中、本地应用程序中或者放在外部程序集中。

引用资源有两种方式: 静态和动态。注意这个区别并不意味着资源本身有任何区别, 这说明不能把资源定义为静态的或动态的。区别在于如何使用它。

1. 静态资源

在设计期间知道使用什么资源, 而且知道该引用不会在应用程序的生存期中变化时, 就使用静态资源。例如, 如果定义了一个按钮样式, 将其用于应用程序中的按钮, 在应用程序运行时就不太可能改变它。此时, 就应静态引用资源。另外, 使用静态资源时, 资源类型在编译期间解析, 所以性能很好。

要引用静态资源, 可以使用下面的标记扩展语法:

```
{StaticResource resourceName}
```

例如，如果为按钮控件定义了一个样式，将其 `x:Key` 特性设置为 `MyStyle`，就可以在控件中引用它，如下所示：

```
<Button Style="{StaticResource MyStyle}" ...>...</Button>
```

## 2. 动态资源

用动态资源定义的属性可以在运行期间改变为另一个动态资源。这在许多情况下都是很有用的。有时希望用户能控制应用程序的一般主题，此时就可以动态分配资源。另外，有时在运行期间并不关心资源需要的键，例如，动态关联了资源程序集。

因此动态资源比静态资源更灵活。但是，动态资源有一个缺点：使用动态资源的系统开销略大，所以如果要优化应用程序的性能，在使用动态资源时应该极其谨慎。

动态引用资源的语法非常类似于静态引用资源的语法：

```
{DynamicResource resourceName}
```

例如，如果为按钮控件定义了一个样式，将其 `x:Key` 特性设置为 `MyDynamicStyle`，就可以在控件中引用它，如下所示：

```
<Button Style="{DynamicResource MyDynamicStyle}" ...>...</Button>
```

## 3. 引用样式资源

前面介绍了如何动态和静态引用按钮控件中的样式资源。这里使用的样式资源位于本地 `Window` 控件的 `Resources` 属性中，例如：

```
<Window ...>
  <Window.Resources>
    <Style x:Key="MyStyle" TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  ...
</Window>
```

每个要使用这个控件样式的按钮控件都必须在其 `Style` 属性中引用它(以静态或动态方式)。另外，还可以定义一个样式资源，使之对给定的控件类型是全局可用的。即 `Style` 对象应用于应用程序中给定类型的每个控件。为此，只需忽略 `x:Key` 特性：

```
<Window ...>
  <Window.Resources>
    <Style TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  ...
</Window>
```

这是给应用程序添加主题的好方法。我们可以为所使用的各种控件类型定义一组全局样式，这样在任意地方都可以使用它们。

前几节讨论了许多基础知识，下面在一个示例中综合使用它们。这个示例修改了上一个示例中的按钮控件，使用触发器和动画，并把样式定义为全局可重用的资源。

### 试一试：触发器、动画和资源

- (1) 创建一个新的 WPF 应用程序 Ch25Ex04，将其保存在 C:\BegVCSharp\Chapter25 目录中。
- (2) 将 Ch25Ex03 的 MainWindow.xaml 代码复制到 Ch25Ex04 的 MainWindow.xaml 中，但修改 Window 元素上的名称空间引用，如下所示：

```
<Window x:Class="Ch25Ex04.MainWindow"
```

代码段 Ch25Ex04\MainWindow.xaml

- (3) 将 Ch25Ex03 的 MainWindow.xaml.cs 中的 Button\_Click() 事件处理程序复制到 Ch25Ex04 的 MainWindow.xaml.cs 中。

- (4) 在<Window>元素上添加一个<Window.Resources>子元素，把<Style>定义从<Button.Style>元素移动到<Window.Resources>元素中。删除空的<Button.Style>元素。结果如下所示(删节)：

```
<Window x:Class="Ch25Ex04.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Nasty Button" Height="150" Width="550">
  <Window.Resources>
    <Style TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  <Grid Background="Black">
    <Button Margin="20" Click="Button_Click">
      Would anyone use a button like this?
    </Button>
  </Grid>
</Window>
```

- (5) 运行应用程序，验证结果与上一个示例相同。
- (6) 在模板的主 Grid 控件和包含 ContentPresenter 元素的 Rectangle 中添加 Name 属性：

```
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="Button">
      <Grid Name="LayoutGrid">
        <Grid.ColumnDefinitions>
          ...
          <Grid Grid.Column="1">
            <Rectangle RadiusX="10" RadiusY="10" Name="BackgroundRectangle">
              <Rectangle.Fill>
                ...
              </Rectangle.Fill>
            </Rectangle>
          </Grid>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
```

```

    ...
</Grid>
</ControlTemplate>
</Setter.Value>
</Setter>

```

(7) 在<ControlTemplate>元素中, 在</ControlTemplate>标记的前面添加如下代码:

```

</Grid>
<ControlTemplate.Resources>
  <Storyboard x:Key="PulseButton">
    <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
      RepeatBehavior="Forever"
      Storyboard.TargetName="BackgroundRectangle"
      Storyboard.TargetProperty=
" (Shape.Fill).(RadialGradientBrush.GradientStops)[1].(GradientStop.Color)">
      <LinearColorKeyFrame Value="Red" KeyTime="00:00:00" />
      <LinearColorKeyFrame Value="Orange" KeyTime="00:00:01" />
      <LinearColorKeyFrame Value="Red" KeyTime="00:00:02" />
    </ColorAnimationUsingKeyFrames>
  </Storyboard>
</ControlTemplate.Resources>
<ControlTemplate.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Setter TargetName="LayoutGrid" Property="Effect">
      <Setter.Value>
        <DropShadowEffect ShadowDepth="0" Color="Red"
          BlurRadius="40" />
      </Setter.Value>
    </Setter>
  </Trigger>
  <Trigger Property="IsPressed" Value="True">
    <Setter TargetName="LayoutGrid" Property="Effect">
      <Setter.Value>
        <DropShadowEffect ShadowDepth="0" Color="Yellow"
          BlurRadius="80" />
      </Setter.Value>
    </Setter>
  </Trigger>
  <EventTrigger RoutedEvent="UIElement.MouseEnter">
    <BeginStoryboard Storyboard="{StaticResource PulseButton}"
      x:Name="PulseButton_BeginStoryboard" />
  </EventTrigger>
  <EventTrigger RoutedEvent="UIElement.MouseLeave">
    <StopStoryboard
      BeginStoryboardName="PulseButton_BeginStoryboard" />
  </EventTrigger>
</ControlTemplate.Triggers>
</ControlTemplate>

```

(8) 运行应用程序, 把鼠标停放在按钮上, 如图 25-16 所示, 按钮会闪动并发光。

(9) 单击按钮, 如图 25-17 所示, 发光效果改变了。

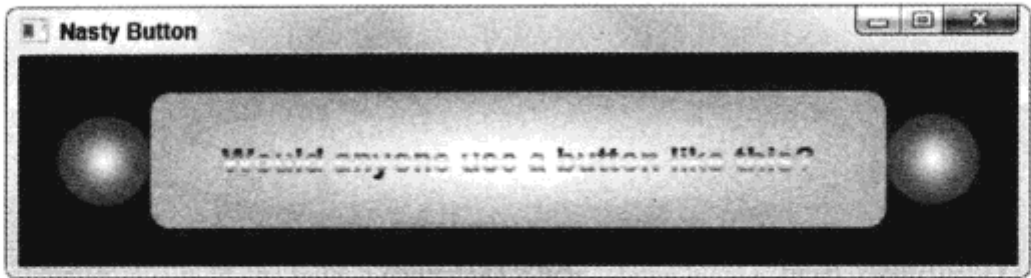


图 25-16

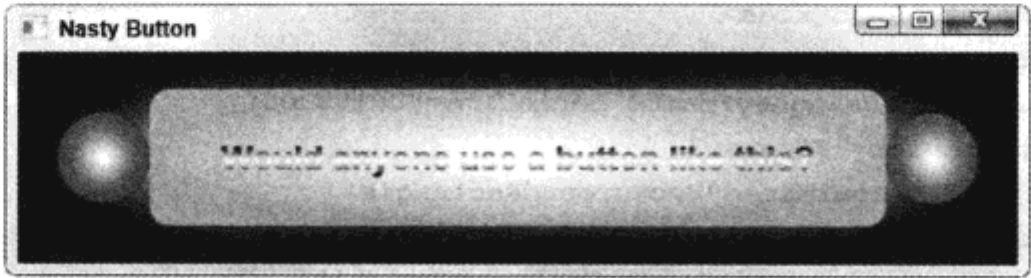


图 25-17

示例的说明

这个示例完成了两个工作。第一，定义了一个全局资源，用于格式化应用程序中的所有按钮(目前只有一个按钮，但这不重要)。第二，给上一个示例创建的样式添加了一些特性，使该样式能基本被接受。该样式会闪动并发光，以响应鼠标停放在按钮上和单击按钮操作。

把样式设置为全局资源，只需把<Style>样式移动到 Window 的资源部分。可以添加 x:Key 特性，但因为这里不需要设置页面上按钮控件的 Style 属性，所以该样式立即可以全局使用。

把样式设置为资源后，还要修改它。首先，给样式中的两个控件添加 Name 特性。这样才能在其他代码中引用它们，本例要为样式的控件模板在动画和触发器中引用它们。

接着，添加一个动画，作为样式中指定的控件模板的一个本地资源。动画 Storyboard 对象用 PulseButton 的 x:Key 值标识：

```
<ControlTemplate.Resources>
  <Storyboard x:Key="PulseButton">
```

故事板包含一个 ColorAnimationUsingKeyFrames 元素，它可以连续改变控件模板中使用的颜色。连续改变的属性是红色，它是在 BackgroundRectangle 控件中使用的放射性填充的外部颜色。在控件中定位这个属性需要为 Storyboard.TargetProperty 关联属性使用相当复杂的语法：

```
<ColorAnimationUsingKeyFrames BeginTime="00:00:00"
  RepeatBehavior="Forever"
  Storyboard.TargetName="BackgroundRectangle"
  Storyboard.TargetProperty=
    "(Shape.Fill).(RadialGradientBrush.GradientStops)[1].(GradientStop.Color)">
```

动画的时间线包含 3 个关键帧，在 2 秒内把颜色从 Red 连续改变为 Orange，再从 Orange 连续改变为 Red：

```
<LinearColorKeyFrame Value="Red" KeyTime="00:00:00" />
<LinearColorKeyFrame Value="Orange" KeyTime="00:00:01" />
<LinearColorKeyFrame Value="Red" KeyTime="00:00:02" />
</ColorAnimationUsingKeyFrames>
</Storyboard>
```

```
</ControlTemplate.Resources>
```

把动画添加为资源，就不会执行动画了。要执行动画，需要添加两个 EventTrigger 触发器：

```
<EventTrigger RoutedEvent="UIElement.MouseEnter">
  <BeginStoryboard Storyboard="{StaticResource PulseButton}"
    x:Name="PulseButton_BeginStoryboard" />
</EventTrigger>
<EventTrigger RoutedEvent="UIElement.MouseLeave">
  <StopStoryboard
    BeginStoryboardName="PulseButton_BeginStoryboard" />
</EventTrigger>
```

这段代码使用 Button 控件的基类 UIElement 中的 MouseEnter 和 MouseLeave 事件，来控制动画的执行。MouseEnter 通过 BeginStoryboard 元素启动动画，MouseLeave 通过 StopStoryboard 元素停止动画。

注意，使用静态资源引用来定位故事板资源。这是很合理的，因为故事板定义为控件的本地资源，不需要在运行期间改变它。

还定义了另外两个触发器，使用 DropShadowEffect 效果产生鼠标停放在按钮上和单击按钮时的发光效果。这里使用本章前面介绍的 IsMouseOver 和 IsPressed 属性来达到这个效果：

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter TargetName="LayoutGrid" Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="0" Color="Red"
        BlurRadius="40" />
    </Setter.Value>
  </Setter>
</Trigger>
<Trigger Property="IsPressed" Value="True">
  <Setter TargetName="LayoutGrid" Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="0" Color="Yellow"
        BlurRadius="80" />
    </Setter.Value>
  </Setter>
</Trigger>
```

鼠标停放在按钮上时，这里定义的发光效果比较小，显示为红色；单击按钮时，该发光效果比较大，显示为黄色。

## 25.4 用 WPF 编程

学习了 WPF 编程的所有基本技术后，就可以开始创建自己的应用程序了。但这里没有足够的篇幅介绍 WPF 的其他优秀特性，如数据绑定、格式化列表显示的一些好方法。但是，熟悉了 WPF 编程后，不应裹足不前，因此在完成本章之前，我们还要介绍两个主题，因为它们不太复杂，而且是 WPF 应用程序中经常执行的任务。

- 如何创建和使用自己的控件



- 如何在控件上实现依赖属性
- 最后还要举一个例子来说明本章介绍的许多技术，并简要讨论 WPF 数据绑定。

25.4.1 WPF 用户控件

WPF 提供了一组在许多情况下都很有用的控件。但是与所有的.NET 开发框架一样，WPF 也允许扩展这个功能。确切地讲，我们可以从 WPF 类层次结构的类中派生自己的类，从而创建自己的控件。

可以派生的一个最有用的控件是 UserControl。这个类提供了 WPF 控件需要的所有基本功能，允许控件无缝地捕捉到已有的 WPF 控件上。希望通过 WPF 控件获得的所有功能，如动画、样式设置、模板化等，都可以通过用户控件获得。

使用 Project | Add User Control 菜单项可以在项目中添加用户控件。这个菜单项会提供一个空白的画布(实际上是一个空白的 Grid)。用户控件在 XAML 中使用顶级的 UserControl 元素定义，代码隐藏类派生于 System.Windows.Controls.UserControl 类。

在项目中添加了用户控件后，就可以添加布局控件和配置控件的代码隐藏内容了。之后，就可以在应用程序中使用用户控件了，甚至可以在其他应用程序中重用它。

创建用户控件时，必须知道的一个重要事项是如何实现依赖属性。如本章前面所述，依赖属性是 WPF 编程的一个重要组成部分。在创建自己的控件时，一般不希望错过这些属性提供的功能。

25.4.2 实现依赖属性

可以在继承自 System.Windows.DependencyObject 类的任何类中添加依赖属性。这个类在 WPF 的许多类的继承层次结构中，包括所有控件和 UserControl。

为了给类实现依赖属性，需要在类型 System.Windows.DependencyProperty 的类定义中添加一个公共静态成员。这个成员的名称由开发人员确定，但最好遵循命名约定<PropertyName> Property:

```
public static DependencyProperty MyStringProperty;
```

把这个属性定义为静态似乎很古怪，因为可以为类的每个实例唯一地定义该属性。WPF 属性框架会自动进行跟踪，所以无需担心它。

必须用静态方法 DependencyProperty.Register()配置刚才添加的成员:

```
public static DependencyProperty MyStringProperty =  
    DependencyProperty.Register(...);
```

这个方法带有 3~5 个参数，如表 25-4 所示(按照顺序，前 3 个参数是必选参数)。

表 25-4

参 数	用 法
string name	属性名
Type propertyType	属性的类型
Type ownerType	包含属性的类的类型
PropertyMetadata typeMetadata	其他属性设置: 属性的默认值, 用于属性改变通知和强制转换的回调方法
ValidateValueCallback	用于验证属性值的回调方法
validateValueCallback	





可以使用其他方法注册依赖属性，如 `RegisterAttached()`，它可以用于实现关联属性。本章不介绍其他方法，但读者应知道它们。

例如，可以使用如下 3 个参数注册 `MyStringProperty` 依赖属性：

```
public class MyClass : DependencyObject
{
    public static DependencyProperty MyStringProperty = DependencyProperty.Register(
        "MyString",
        typeof(string),
        typeof(MyClass));
}
```

也可以包含能直接访问依赖属性的 .NET 属性(但这不是强制的，如后面所示)。但是，因为依赖属性定义为静态成员，所以不能对它们使用与普通属性相同的语法。要访问依赖属性的值，必须使用从 `DependencyObject` 继承的方法，如下所示：

```
public class MyClass : DependencyObject
{
    public static DependencyProperty MyStringProperty = DependencyProperty.Register(
        "MyString",
        typeof(string),
        typeof(MyClass));

    public string MyString
    {
        get { return (string)GetValue(MyStringProperty); }
        set { SetValue(MyStringProperty, value); }
    }
}
```

其中 `GetValue()` 和 `SetValue()` 方法分别获取和设置当前实例的 `MyStringProperty` 依赖属性值。这两个方法都是公共的，所以客户代码可以直接使用它们处理依赖属性的值。因此添加一个 .NET 属性来访问依赖属性并非是强制的。

如果要为属性设置元数据，就必须使用派生于 `PropertyMetadata` 的对象，如 `FrameworkPropertyMetadata`，把这个实例作为第四个参数传送给 `Register()`。`FrameworkPropertyMetadata` 构造函数有 11 个重载版本，它们带有一个或多个参数，如表 25-5 所示。

表 25-5

属性类型	用法
object defaultValue	属性的默认值
FrameworkPropertyMetadataOptionsflags	标记的组合(来自于 <code>FrameworkPropertyMetadataOptions</code> 枚举)，用于指定属性的其他元数据。例如，使用 <code>AffectArrange</code> 可以声明对属性的改变会影响控件的布局。这会使窗口的布局引擎在属性改变时重新计算控件的布局。MSDN 文档中列出了这个选项的完整列表

(续表)

属性类型	用法
propertyChangedCallback propertyChangedCallback	属性值改变时使用的回调方法
CoerceValueCallback coerceValueCallback	属性值强制转换时使用的回调方法
bool isAnimationProhibited	指定这个属性是否被动画改变
UpdateSourceTrigger defaultUpdateSourceTrigger	对属性值进行数据绑定操作时，这个属性根据枚举 UpdateSourceTrigger 中的值确定何时更新数据源。默认值是 PropertyChanged，表示只要属性改变，就更新绑定源。这未必合适，例如，TextBox.Text 属性给这个属性使用 LostFocus 值，确保绑定源并不过早更新。还可以使用 Explicit 值指定绑定源仅在请求时更新(调用派生于 DependencyObject 的类的 UpdateSource()方法)

下面是使用 FrameworkPropertyMetadata 的一个简单示例，它只设置了属性的默认值：

```
public class MyClass : DependencyObject
{
    public static DependencyProperty MyStringProperty =
        DependencyProperty.Register(
            "MyString",
            typeof(string),
            typeof(MyClass),
            new FrameworkPropertyMetadata("Default value"));
}
```

前面学习了可以指定的 3 个回调方法，分别用于属性改变通知、属性强制转换和属性值的验证。这些回调方法与依赖属性一样，必须实现为公共静态方法。每个回调方法都有一个指定的返回类型和参数列表。

下面的示例要创建并使用一个用户控件，它有两个依赖属性。我们将学习如何在用户控件代码中为这些属性实现回调方法。

试一试：用户控件

- (1) 创建一个新的 WPF 应用程序 Ch25Ex05，将其保存在 C:\BegVCSharp\Chapter25 目录中。
- (2) 在应用程序中添加一个新的用户控件 Card，修改 Card.xaml 中的代码，如下所示：



```
<UserControl x:Class="Ch25Ex05.Card"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="150" d:DesignWidth="100"
    Height="150" Width="100" x:Name="UserControl"
    FontSize="16" FontWeight="Bold">
<UserControl.Resources>
```

```

<DataTemplate x:Key="SuitTemplate">
    <TextBlock Text="{Binding}" />
</DataTemplate>
</UserControl.Resources>
<Grid>
    <Rectangle Stroke="{x:Null}" RadiusX="12.5" RadiusY="12.5">
        <Rectangle.Fill>
            <LinearGradientBrush EndPoint="0.47,-0.167" StartPoint="0.86,0.92">
                <GradientStop Color="#FFD1C78F" Offset="0" />
                <GradientStop Color="#FFFFFF" Offset="1" />
            </LinearGradientBrush>
        </Rectangle.Fill>
        <Rectangle.Effect>
            <DropShadowEffect/>
        </Rectangle.Effect>
    </Rectangle>
    <Label x:Name="SuitLabel"
        Content="{Binding Path=Suit, ElementName=UserControl, Mode=Default}"
        ContentTemplate="{DynamicResource SuitTemplate}"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Margin="8,51,8,60" />
    <Label x:Name="RankLabel"
        Content="{Binding Path=Rank, ElementName=UserControl, Mode=Default}"
        ContentTemplate="{DynamicResource SuitTemplate}"
        HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="8,8,0,0" />
    <Label x:Name="RankLabelInverted"
        Content="{Binding Path=Rank, ElementName=UserControl, Mode=Default}"
        ContentTemplate="{DynamicResource SuitTemplate}"
        HorizontalAlignment="Right" VerticalAlignment="Bottom"
        Margin="0,0,8,8" RenderTransformOrigin="0.5,0.5">
        <Label.RenderTransform>
            <RotateTransform Angle="180" />
        </Label.RenderTransform>
    </Label>
    <Path Fill="#FFFFFF" Stretch="Fill" Stroke="{x:Null}"
        Margin="0,0,35.218,-0.077" Data="F1 M10.5,51 L123.16457,51 C116.5986,
        76.731148 115.63518,132.69684 121.63533,149.34013 133.45299,
        182.12018 152.15821,195.69803 161.79765,200.07669 L110.5,200 C103.59644,
        200 98,194.40356 98,187.5 L98,63.5 C98,56.596439 103.59644,51 110.5,51 z">
        <Path.OpacityMask>
            <LinearGradientBrush EndPoint="0.957,1.127" StartPoint="0,-0.06">
                <GradientStop Color="#FF000000" Offset="0" />
                <GradientStop Color="#00FFFFFF" Offset="1" />
            </LinearGradientBrush>
        </Path.OpacityMask>
    </Path>
</Grid>
</UserControl>

```

代码段 Ch25Ex05\Card.xaml

(3) 修改 Card.xaml.cs 中的代码, 如下所示:

```
public partial class Card : UserControl
{
    public static string[] Suits = { "Club", "Diamond", "Heart", "Spade" };

    public static DependencyProperty SuitProperty = DependencyProperty.Register(
        "Suit",
        typeof(string),
        typeof(Card),
        new PropertyMetadata("Club", new PropertyChangedCallback(OnSuitChanged)),
        new ValidateValueCallback(ValidateSuit));

    public static DependencyProperty RankProperty = DependencyProperty.Register(
        "Rank",
        typeof(int),
        typeof(Card),
        new PropertyMetadata(1),
        new ValidateValueCallback(ValidateRank));

    public Card()
    {
        InitializeComponent();
    }

    public string Suit
    {
        get { return (string)GetValue(SuitProperty); }
        set { SetValue(SuitProperty, value); }
    }

    public int Rank
    {
        get { return (int)GetValue(RankProperty); }
        set { SetValue(RankProperty, value); }
    }

    public static bool ValidateSuit(object suitValue)
    {
        string suitValueString = (string)suitValue;
        if (suitValueString != "Club" && suitValueString != "Diamond"
            && suitValueString != "Heart" && suitValueString != "Spade")
        {
            return false;
        }
        return true;
    }

    public static bool ValidateRank(object rankValue)
    {
        int rankValueInt = (int)rankValue;
        if (rankValueInt < 1 || rankValueInt > 12)
        {
            return false;
        }
    }
}
```

```

        return true;
    }

    private void SetTextColor()
    {
        if (Suit == "Club" || Suit == "Spade")
        {
            RankLabel.Foreground = new SolidColorBrush(Color.FromRgb(0, 0, 0));
            SuitLabel.Foreground = new SolidColorBrush(Color.FromRgb(0, 0, 0));
            RankLabelInverted.Foreground =
                new SolidColorBrush(Color.FromRgb(0, 0, 0));
        }
        else
        {
            RankLabel.Foreground = new SolidColorBrush(Color.FromRgb(255, 0, 0));
            SuitLabel.Foreground = new SolidColorBrush(Color.FromRgb(255, 0, 0));
            RankLabelInverted.Foreground =
                new SolidColorBrush(Color.FromRgb(255, 0, 0));
        }
    }

    public static void OnSuitChanged(DependencyObject source,
        DependencyPropertyChangedEventArgs args)
    {
        ((Card) source).SetTextColor();
    }
}

```

代码段 Ch25Ex05\Card.xaml.cs

(4) 修改 MainWindow.xaml 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

<Window x:Class="Ch25Ex05.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Card Dealer" Height="600" Width="800">

    <Grid Name="contentGrid" MouseLeftButtonDown="Grid_MouseLeftButtonDown"
        MouseLeftButtonUp="Grid_MouseLeftButtonUp" MouseMove="Grid_MouseMove">
        <Grid.Background>
            <LinearGradientBrush EndPoint="0.364,0.128" StartPoint="0.598,1.042">
                <GradientStop Color="#FF0D4F1A" Offset="0"/>
                <GradientStop Color="#FF448251" Offset="1"/>
            </LinearGradientBrush>
        </Grid.Background>
    </Grid>
</Window>

```

代码段 Ch25Ex05\MainWindow.xaml

(5) 修改 MainWindow.xaml.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```

public partial class MainWindow : Window
{
    private Card currentCard;
    private Point offset;
}

```

```

private Random random = new Random();

public MainWindow()
{
    InitializeComponent();
}

private void Grid_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (e.Source is Card)
    {
        currentCard = (Card)e.Source;
        offset = Mouse.GetPosition(currentCard);
        contentGrid.Children.Remove(currentCard);
    }
    else
    {
        currentCard = new Card
        {
            Suit = Card.Suits[random.Next(0, 4)],
            Rank = random.Next(1, 13)
        };
        currentCard.HorizontalAlignment = HorizontalAlignment.Left;
        currentCard.VerticalAlignment = VerticalAlignment.Top;
        offset = new Point(50, 75);
    }
    contentGrid.Children.Add(currentCard);
    PositionCard();
}

private void Grid_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    currentCard = null;
}

private void Grid_MouseMove(object sender, MouseEventArgs e)
{
    if (currentCard != null)
    {
        PositionCard();
    }
}

private void PositionCard()
{
    Point mousePos = Mouse.GetPosition(this);
    currentCard.Margin =
        new Thickness(mousePos.X - offset.X, mousePos.Y - offset.Y, 0, 0);
}
}

```

代码段 Ch25Ex05\MainWindow.xaml.cs

(6) 运行应用程序。单击窗口表面，会添加随机扑克牌，单击并拖动可以重新定位扑克牌。单

击已有的扑克牌时，它会跳到堆栈顺序的顶部。结果如图 25-18 所示。

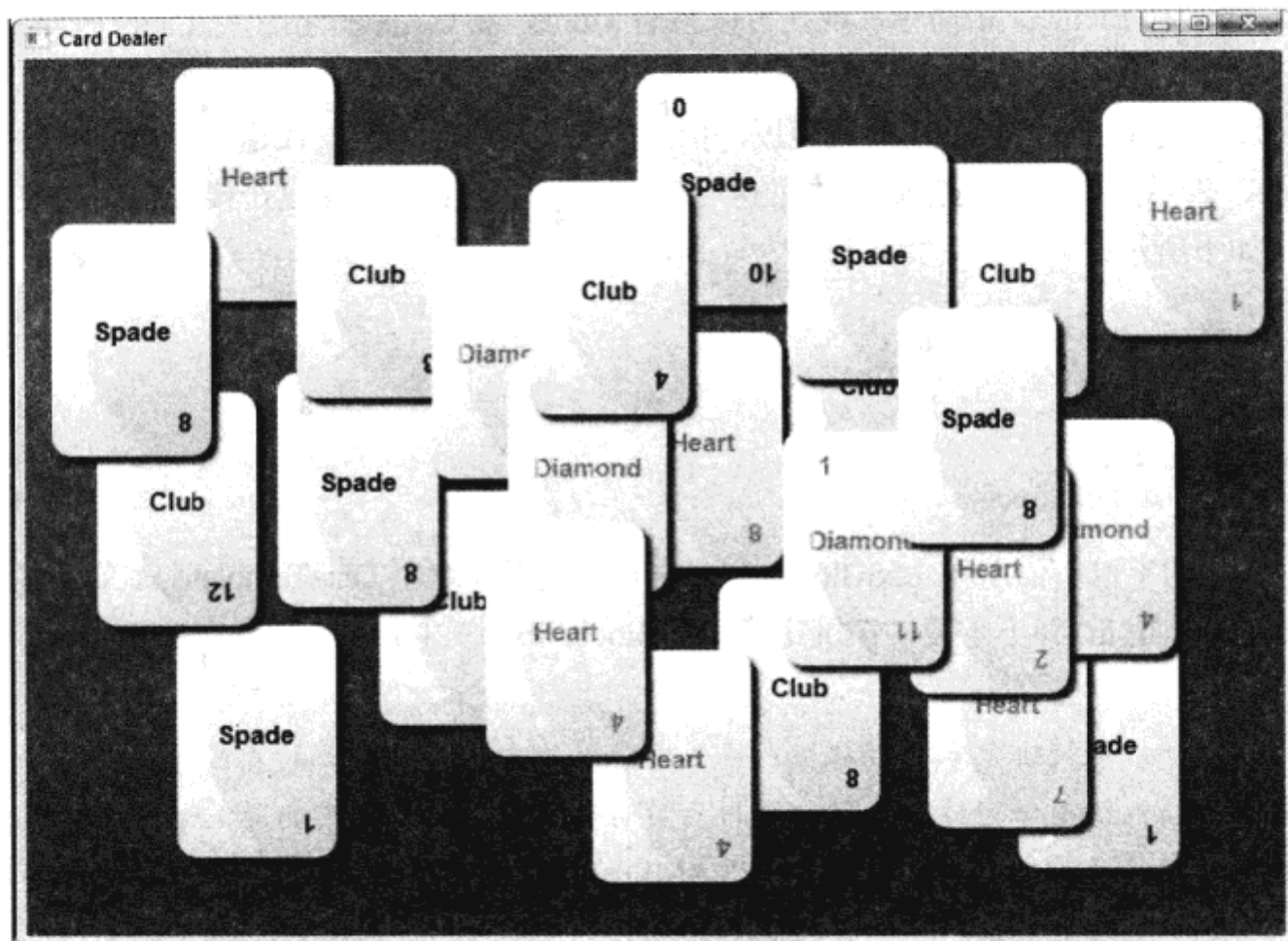


图 25-18

### 示例的说明

这个示例创建了一个用户控件，它带有两个依赖属性。该示例包含的客户代码使用这个控件。这个示例包含许多基础知识，首先要在代码中查看的是 Card 控件。

Card 控件包含的大多数代码都在本章前面使用过了。布局代码没有使用新特性，但结果比前面两个例子的不很美观按钮好多了。有一个全新的特性：这个控件使用了一小部分数据绑定功能。在使用数据绑定操作时，控件的属性绑定到数据源上，其中包含许多技术。WPF 很容易把属性绑定到所有的数据源上，如数据库中的数据、XML 数据和依赖属性值(本例就使用了依赖属性值)。

确切地讲，Card 中的代码为客户代码提供了两个依赖属性：Suit 和 Rank，并把这两个属性绑定到控件布局中的可视化元素上。结果是把 Suit 设置为 Club 时，单词 Club 就显示在扑克牌的中心。同样，Rank 的值显示在扑克牌的两个角上。

稍后讨论 Suit 和 Rank 的实现代码。现在只要了解这些属性分别是 string 和 int 值就足够了。可以给这两个属性使用枚举值，但这需要编写不少新代码，所以这个示例使用基本属性，以使代码尽可能简单。

要把一个值绑定到属性上，应使用绑定语法，即标记扩展。这个语法表示，把属性值指定为 {Binding...}。以这种方式配置绑定有许多可能性。在本示例中，SuitLabel 标签的绑定配置为：

```
<Label x:Name="SuitLabel"
  Content="{Binding Path=Suit, ElementName=UserControl, Mode=Default}"
  ContentTemplate="{DynamicResource SuitTemplate}" HorizontalAlignment="Center"
  VerticalAlignment="Center" Margin="8,51,8,60"/>
```



这里为绑定指定了 3 个属性: Path(属性名)、ElementName(与属性关联的元素)和 Mode(执行绑定的方式)。Path 和 Element 很简单, 现在可以忽略 Mode, 要点是这个指定把 Label.Content 属性绑定到 Card.Suit 属性上。

在绑定属性值时, 还必须使用数据模板, 指定如何显示绑定内容。在这个示例中, 数据模板是 SuitTemplate, 表示为一个动态资源(但这里也可以使用静态资源绑定)。这个模板在用户控件资源部分中定义, 如下所示:

```
<UserControl.Resources>
  <DataTemplate x:Key="SuitTemplate">
    <TextBlock Text="{Binding}" />
  </DataTemplate>
</UserControl.Resources>
```

因此, Suit 的字符串值用作 TextBlock 控件的 Text 属性。这个 DataTemplate 定义也重用于两个 Rank 标签。Rank 是 int 并不重要, 在绑定到 TextBlock.Text 属性上时, 它会转换为 string。



显然, 数据绑定和数据模板有许多内容, 但本书没有足够的篇幅介绍其细节。本章的小结仅给出学习这个主题的资料。最后要注意, 如果使用的是 Expression Blend, 就可以高效地绑定数据, 无需考虑 XAML 语法, 因为 Expression Blend 会处理语法。

为了使这个数据绑定起作用, 必须使用上一节介绍的技术定义两个依赖属性。它们在用户控件的代码隐藏内容中定义, 如下所示(它们都是简单的.NET 属性包装器, 所以不需要显示其代码):

```
public static DependencyProperty SuitProperty =
    DependencyProperty.Register(
        "Suit",
        typeof(string),
        typeof(Card),
        new PropertyMetadata(
            "Club", new PropertyChangedCallback(OnSuitChanged)),
        new ValidateValueCallback(ValidateSuit));

public static DependencyProperty RankProperty =
    DependencyProperty.Register(
        "Rank",
        typeof(int),
        typeof(Card),
        new PropertyMetadata(1),
        new ValidateValueCallback(ValidateRank));
```

两个依赖属性都使用回调方法验证其值, Suit 属性还有一个在其值改变时执行的回调方法。验证回调方法的返回类型是 bool, 它只有一个 object 类型的参数, 它是客户代码试图设置的属性值。如果该值是 OK, 就应返回 true, 否则就返回 false。在示例代码中, 只允许 Suit 属性使用以下 4 个字符串中的一个:

```
public static bool ValidateSuit(object suitValue)
{
    string suitValueString = (string)suitValue;
```

```

    if (suitValueString != "Club"&& suitValueString != "Diamond"
        && suitValueString != "Heart"&& suitValueString != "Spade")
    {
        return false;
    }
    return true;
}

```

这是相当麻烦的，显然这里使用枚举比较好，但没有这么做，原因如前所述。同样，Rank 属性也被限制为 1(ace)~12(king)之间：

```

public static bool ValidateRank(object rankValue)
{
    int rankValueInt = (int)rankValue;
    if (rankValueInt < 1 || rankValueInt > 12)
    {
        return false;
    }
    return true;
}

```

Suit 的值改变时，就调用 OnSuitChanged()回调方法。这个方法负责把文本颜色设置为红色(表示红心和方块)或黑色(表示黑桃和梅花)。为此，在方法调用的源上调用一个实用方法。这是必须的，因为回调方法实现为静态方法，但要把引发事件的用户控件实例传送为一个参数，才能与用户控件交互操作。所调用的方法是 SetTextColor()：

```

public static void OnSuitChanged(DependencyObject source,
    DependencyPropertyChangedEventArgs args)
{
    ((Card)source).SetTextColor();
}

```

SetTextColor()方法是私有方法，但显然仍可在 OnSuitChanged()中访问，因为它们都是同一个类的成员，分别是实例和静态方法。SetTextColor()仅根据 Suit 值把控件的各个标签的 Foreground 属性设置为黑色或红色的纯色笔刷：

```

private void SetTextColor()
{
    if (Suit == "Club" || Suit == "Spade")
    {
        RankLabel.Foreground =
            new SolidColorBrush(Color.FromRgb(0, 0, 0));
        SuitLabel.Foreground =
            new SolidColorBrush(Color.FromRgb(0, 0, 0));
        RankLabelInverted.Foreground =
            new SolidColorBrush(Color.FromRgb(0, 0, 0));
    }
    else
    {
        RankLabel.Foreground =
            new SolidColorBrush(Color.FromRgb(255, 0, 0));
        SuitLabel.Foreground =

```

```
        new SolidColorBrush(Color.FromRgb(255, 0, 0));  
        RankLabelInverted.Foreground =  
            new SolidColorBrush(Color.FromRgb(255, 0, 0));  
    }  
}
```

这就是 Card 控件中需要查看的代码。MainWindow.xaml 和 MainWindow.xaml.cs 中的客户代码相当简单,它使用基本的样式提供渐变的绿色背景,还提供了许多事件处理代码(使用路由事件和关联路由事件)进行用户交互操作。其中有几个难点——例如,页边距如何用于定位扑克牌,如何使用偏移值,把已有的扑克牌从单击的位置拖动开——阅读这些代码,就可以攻克这些难点。

## 25.5 小结

本章学习了用 WPF 编程所需的所有基础知识,还简要讨论了一些高级技术,了解了高级 WPF 编程提供的功能。WPF 主题的范围极广,不可能用一章的篇幅详细论述,如果读者对 WPF 感兴趣,可以参阅这个主题的其他资料,例如,如果希望更多地了解 Web 环境下的 XAML,开始时可以参考 *WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4* (Wrox, 2010)或者 *Silverlight 3 Programmer's Reference* (Wrox, 2009)。Silverlight 是一个尤其需要关注的领域,它的功能变得越来越好了。由于 Silverlight 是 Web 开发环境下的 WPF 的一个子集,所以注意 WPF 和 Silverlight 技巧很多都是通用的。

当然,也可以使用可用的工具(尤其是 Expression Blend)看看能得到什么效果。MSDN 文档也有一定的帮助,只是 WPF 是一种新技术,这方面的资料还比较欠缺,必须到其他地方查找相关信息。

有一些很好的网站提供了额外的信息。例如 WPF 的社区站点 <http://windowsclient.net/>,还可以访问 Scott Guthrie 的博客 <http://weblogs.asp.net/scottgu>。

本章介绍的内容如下:

- WPF 的含义,它对桌面和 Web 软件开发的潜在影响
- WPF 如何让使用 Expression Blend 的设计人员和使用 VS 或 VCE 的开发人员协同工作,完成项目
- XAML 的含义,它的基本语法和一些术语
- 使用 Application 对象
- WPF 中的控件的工作原理,包括依赖属性、关联属性、路由事件和关联事件的概念
- WPF 中的布局系统,使用各种布局容器定位控件
- 用样式和模板定制控件的外观和操作方式
- 使用触发器和动画改进用户体验
- 在内部和外部资源字典中定义资源,静态和动态访问资源
- 创建带依赖属性的用户控件

第 26 章将介绍 .NET 3.0 最近引入的另一个技术: Windows Communication Foundation。

25.6 练习

- (1) 有人说，可以给 WPF 桌面应用程序和 WPF 浏览器应用程序使用相同的 XAML 代码，这种说法正确吗？
- (2) 为了允许子控件给父控件上定义的属性设置值，需要使用什么技术？应在 XAML 中使用什么语法？给出一个 XAML 示例，其中有两个子控件 Branch 为父控件 Tree 上定义的属性 LeafCount 设置不同的值。
- (3) 下面哪个关于依赖属性的论述是正确的？
- a. 必须通过关联的.NET 属性来访问依赖属性。
  - b. 依赖属性定义为公共静态成员。
  - c. 每个类定义只能有一个依赖属性。
  - d. 必须用命名约定[PropertyName]Property 来命名依赖属性。
  - e. 可以用回调方法验证赋给依赖属性的值。
- (4) 可以使用哪个布局控件在单行或单列中显示控件？
- (5) WPF 中的通道事件以特殊的方式命名，以便识别它们。这个命名约定是什么？
- (6) 哪些属性类型可以连续改变？
- (7) 何时使用动态资源引用，而不是静态资源引用？
- 附录 A 给出了练习答案。

25.7 本章要点

主 题	重 要 概 念
WPF 的含义	WPF 是 Microsoft 创建 Windows 和 Web 应用程序的最新方式，它使用标记和代码隐藏模型，能清晰地分开设计和功能，当设计人员和开发人员必须同时处理一个项目时，这种技术非常有效。WPF 中的标记称为 XAML。设计人员常使用 Expression Blend 处理 XAML
XAML	XAML 的语法允许在标记中表示对象。XAML 用 XML 编写，使用标记扩展来提供 WPF 功能
控件	WPF 使用控件建立用户界面，非常类似于 Windows 窗体。WPF 控件使用依赖属性提供与框架各个方面的集成，例如更改通知。依赖属性可以关联到未定义它们的对象上，并在需要时提供上下文信息。WPF 中的控件事件常常是路由事件，它们会沿着 WPF 应用程序的控件层次结构向上或向下路由
布局	WPF 提供了几个布局控件，它们可以包含其他控件。根据所需布局的类型，可以使用 Canvas、DockPanel、Grid、StackPanel 或 WrapPanel 布局控件
样式	可以使用 Style 对象给控件指定样式，Style 对象主要包含应用于控件属性的 Setter 对象。通过修改其模板，可以完全控制控件的外观和操作方式，还可以使用触发器响应用户的交互操作和数据变化
动画	在某个时间段内，属性可以在一个取值范围内连续变化。使用关键帧可以定义动画中的重点，通过编程方式或触发器启动和停止动画
资源	可以为任意作用域定义资源(尤其是样式和模板)，例如在控件、窗口或应用程序的范围内定义。可以使用 StaticResource 或 DynamicResource 标记扩展访问资源，这取决于引用是否需要随时间变化



# 第 26 章

## Windows Communication Foundation

### 本章内容:

---

- WCF 的含义
- WCF 概念
- WCF 编程

第 19 章学习了 Web 服务, 以及如何使用它们在应用程序之间提供简单的通信。该章探讨了如何使用 HTTP GET 和 POST 技术与 Web 服务交换数据, 以及如何使用 SOAP。自从 Web 服务可供 .NET 开发人员使用以来, 尽管 Web 服务很强大, 但这种技术的扩展显然是有一定限制的。为此, 微软公司发布了 Web Service Enhancements(WSE)插件来解决这个问题。WSE 允许 Web 服务开发人员通过消息的安全保护、路由技术和各种其他策略来提高 Web 服务, 但仍有进一步提升的空间。

另一种 .NET 技术——远程技术可以在一个进程中创建对象实例, 在另一个进程中使用它们。远程技术甚至允许在一台计算机上使用位于另一台计算机上的对象。但这种技术仍有它自己的问题。远程技术是有限制的, 刚入门的程序员要掌握它也不容易。

Windows Communication Foundation(WCF)是 Web 服务和远程技术的替代品, 它从 Web 服务中提取了服务、独立于平台的 SOAP 消息传输等概念, 把它们与远程技术中的宿主服务器应用程序和高级绑定功能结合在一起, 所以可以将这种技术看作一个超集, 包含了 Web 服务和远程技术, 但比 Web 服务强大, 比远程技术更易于掌握。使用 WCF 可以从简单的应用程序转向使用面向服务的体系结构(SOA)的应用程序。SOA 表示可以分散处理, 并在需要时连接跨本地网络和 Internet 的服务和数据, 使用分布式处理。

本章将学习 WCF 的原理, 以及如何在应用程序代码中创建和使用 WCF 服务。





不能在 VCE(Visual C# 2010 Express)中创建 WCF 服务,但可以在 VS 的完全版本中创建。还可以在 Visual Web Developer 2010 Express 中创建 IIS 承载的 WCF 服务,但本章使用 VS 来介绍所有的选项。

## 26.1 WCF 的含义

WCF 技术允许创建服务,可以跨进程、计算机和网络从其他应用程序访问这些服务。利用这些服务,可以共享跨多个应用程序的功能,提供数据源,或者抽象复杂的进程。

与 Web 服务一样, WCF 服务提供的功能也封装为该服务的方法。每个方法——在 WCF 术语中称为“操作(operation)”——都有一个端点,用于交换数据。在这一点上, WCF 与 Web 服务不同。在 Web 服务中,只能在 HTTP 上通过 SOAP 与端点通信。而在 WCF 服务中,可以选择要使用的协议。甚至可以通过多个协议与端点通信,这取决于通过什么网络连接服务和特定的要求。

在 WCF 上,端点可以有多个绑定,每个绑定都指定了一种通信方式。绑定还可以指定其他信息,例如,必须满足什么安全要求才能与端点通信。例如,绑定可能需要用户名和密码身份验证或者 Windows 用户账户令牌。在连接一个端点时,绑定使用的协议会影响所使用的地址,如后面所述。

一旦连接了一个端点,就可以使用 SOAP 消息与它通信。所使用的消息形式取决于所进行的操作和该操作收发消息所需的数据结构。WCF 使用合同(contract)指定所有这些信息。通过与服务交换的元数据可以查找合同。这类似于 Web 服务使用 WSDL 描述其功能。实际上,可以用 WSDL 格式获得 WCF 服务的相关信息,但 WCF 服务还可以用其他方式描述。

识别出要使用的服务和端点,知道了要使用的绑定和需要依从的合同之后,就可以与 WCF 服务通信,这与使用在本地定义的对象一样简单。与 WCF 服务通信可以是简单的单向事务、请求/响应消息,也可以是从通信信道任一端发出的全双工通信。还可以在需要时使用消息负载优化技术,如 Message Transmission Optimization Mechanism(MTOM)来打包数据。

WCF 服务在存储它的计算机上运行为许多不同进程中的一个。Web 服务总是运行在 IIS 上,而 WCF 服务可以选择适合的宿主进程。可以使用 IIS 驻留 WCF 服务,也可以使用 Windows 服务或可执行程序。如果使用 TCP 在本地网络上与 WCF 服务通信,就不需要在运行服务的 PC 上安装 IIS。

WCF 架构允许定制本节介绍的几乎所有方面。但这是一个高级主题,本章仅使用 .NET 4 默认提供的技术。

了解了 WCF 服务的基础知识后,下面将详细介绍这些概念。

## 26.2 WCF 概念

本节描述 WCF 的如下方面:

- WCF 通信协议
- 地址、端点和绑定
- 合同



- 消息模式
- 行为
- 驻留

26.2.1 WCF 通信协议

- 如前所述，可以通过许多传输协议与 WCF 服务通信。在.NET 4 Framework 中定义了 4 个协议：
- **HTTP**：它允许与任何地方(包括跨 Internet)的 WCF 服务通信。可以使用 HTTP 通信技术创建 WCF Web 服务。
  - **TCP**：如果正确配置了防火墙，它允许与本地网络或跨 Internet 的 WCF 服务通信。TCP 比 HTTP 高效，功能也比较多，但配置起来更加复杂。
  - **命名管道**：它允许与 WCF 服务通信，该 WCF 服务与调用代码位于同一台计算机的不同进程上。
  - **MSMQ**：这是一种排队技术，允许应用程序发送的消息通过队列路由到目的地。MSMQ 是一种可靠的消息传输技术，可以确保发送给队列的消息一定达到该队列。MSMQ 还是一种异步技术，所以只有排在前面的消息都处理完了，服务仍有效时，才能处理当前的消息。

这些协议常常允许建立安全连接。例如，可以使用 HTTPS 协议建立 Internet 上的安全 SSL 连接。TCP 使用 Windows 安全架构为本地网络上的安全性能提供了更多的可能性。

图 26-1 列出了这些传输协议如何把应用程序与不同位置上的 WCF 服务连接起来。本章将介绍所有这些协议，但 MSMQ 除外(这个主题需要较深入的讨论)。

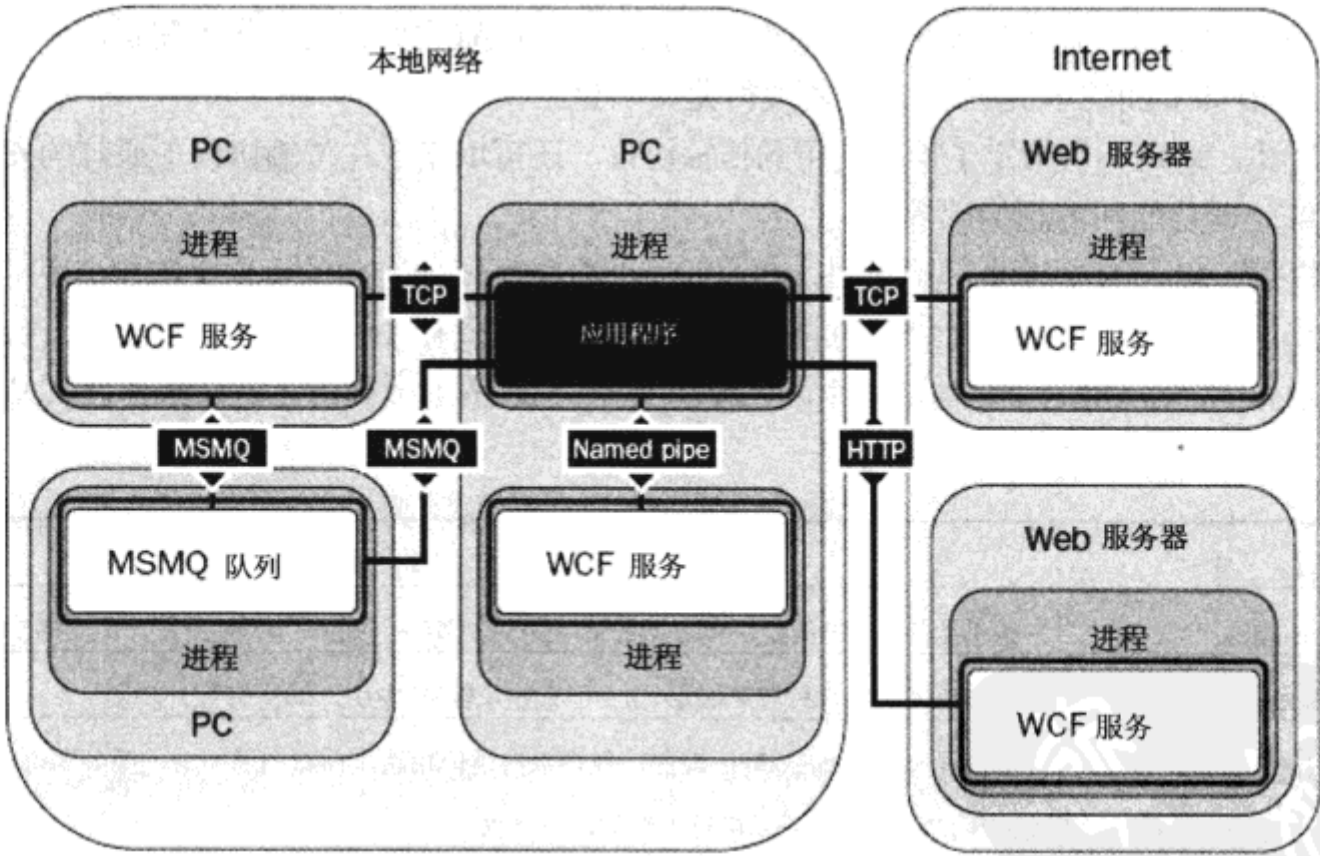


图 26-1

为了连接 WCF 服务，必须知道它在什么地方。这表示必须知道端点的地址。

26.2.2 地址、端点和绑定

用于服务的地址类型取决于所使用的协议。本章前面介绍的 3 个协议(不包括 MSMQ)都需要格

式化的服务地址：

- HTTP：HTTP 协议的地址是 URL，其格式很常见：`http://<server>:<port>/<service>`。对于 SSL 连接，也可以使用 `https://<server>:<port>/<service>`。如果在 IIS 中承载服务，<service> 就是扩展名为.svc 的文件(.svc 文件类似于 Web 服务中使用的.asmx 文件)。IIS 地址可能包含比这个示例更多的子目录，即.svc 文件之前有更多使用/字符分隔的部分。
- TCP：TCP 的地址采用 `net.tcp:// <server>:<port>/<service>`形式。
- 命名管道：命名管道连接的地址与上述类似，但没有端口号。其形式是 `net.pipe:// <server>/<service>`。

服务的地址是一个基地址，它可用于为表示操作的端点创建地址。例如，在 `net.tcp://<server>:<port>/<service>/operation1` 上有一个操作。

假定创建一个 WCF 服务，它有一个操作，绑定了前面介绍的3个协议，我们就可以使用下面的基地址：

```
http://www.mydomain.com/services/amazingservices/mygreatservice.svc
net.tcp://myhugeserver:8080/mygreatservice
net.pipe://localhost/mygreatservice
```

接着就可以给操作使用下面的地址：

```
http://www.mydomain.com/services/amazingservices/mygreatservice.svc/greatop
net.tcp://myhugeserver:8080/mygreatservice/greatop
net.pipe://localhost/mygreatservice/greatop
```

在.NET 4 中，可以给操作使用默认端点，而无需明确配置它们。这简化了配置，如果需要使用标准端点地址(如上面的示例所示)，这表现得尤其明显。

如前所述，绑定不仅指定了操作使用的传输协议，还可以指定在传输协议上通信的安全要求、端点的事务处理功能和消息编码等。

绑定提供了极大的灵活性，所以.NET Framework 提供了一些可用的预定义绑定。还可以把这些绑定用作起点，修改它们，得到需要的绑定类型。预定义绑定有一些必须遵循的原则。每种绑定类型都用 System.ServiceModel 名称空间中的一个类表示。表 26-1 中列出了这些绑定及其基本信息。

表 26-1

绑 定	说 明
BasicHttpBinding	最简单的 HTTP 绑定，Web 服务使用的默认绑定，它的安全功能有限，不支持事务处理
WSHttpBinding	HTTP 绑定的一种较高级形式，可以使用 WSE 中引入的所有额外功能
WSDualHttpBinding	扩展了 WSHttpBinding 功能，包含双向通信功能。在双向通信中，服务器可以启动与客户机的通信，还可以进行一般的消息交换
WSFederationHttpBinding	扩展了 WSHttpBinding 功能，包含联合功能。联合功能允许第三方实现单一登录(single sign-on)和其他专用安全措施。这是一个高级主题，本章不予讨论
NetTcpBinding	用于 TCP 通信，允许配置安全性、事务处理等
NetNamedPipeBinding	用于指定管道的通信，允许配置安全性、事务处理等
NetPeerTcpBinding	允许与多个客户机进行广播通信，是一个高级类，本章不予讨论

(续表)

绑 定	说 明
NetMsmqBinding 和 MsmqIntegrationBinding	这些绑定用于 MSMQ，本章不予讨论
NetPeerTcpBinding	用于对等绑定，本章不予讨论
WebHttpBinding	用于使用 HTTP 请求(而不是 SOAP 消息)的 Web 服务
NetTcpContextBinding	类似于 NetTcpBinding，但允许通过 SOAP 标题交换环境信息
BasicHttpContextBinding 和 WSHttpContextBinding	类似于 BasicHttpBinding 和 WSHttpBinding，但允许分别通过 HTTP Cookie 或 SOAP 标题交换环境信息

这个表中的许多绑定类拥有可用于其他配置的类似属性。例如，它们有可用于配置超时值的属性。本章后面介绍编码时会详细讨论。

在.NET 4 中，端点的默认绑定因所用协议而异。这些默认绑定如表 26-2 所示。

表 26-2

协 议	默 认 绑 定
HTTP	BasicHttpBinding
TCP	NetTcpBinding
命名管道	NetNamedPipeBinding
MSMQ	NetMsmqBinding

26.2.3 合同

合同定义了 WCF 服务的用法。可以定义如下几种合同：

- **服务合同**：包含服务的一般信息和服务提供的操作的一般信息。例如，该合同可以包含服务使用的名称空间。在为 SOAP 消息定义模式时，服务使用唯一的名称空间，以免与其他服务冲突。
- **操作合同**：定义操作的用法，这包括操作方法的参数和返回类型，以及其他信息，例如，方法是否返回响应消息。
- **消息合同**：允许定制 SOAP 消息内部的信息格式化方式。例如，数据应包含在 SOAP 标题中还是 SOAP 消息体中。在创建必须与以前的系统集成的 WCF 服务时，就可以使用消息合同。
- **错误合同**：定义操作可能返回的错误。使用.NET 客户程序时，错误会导致可以捕获的异常，并以通常方式处理。
- **数据合同**：如果使用复杂类型，如用户定义的结构和对象(作为操作的参数或返回类型)，就必须为这些类型定义数据合同。数据合同根据通过属性显示的数据来定义类型。

一般使用特性把合同添加到服务类和方法中，如本章后面所述。

26.2.4 消息模式

上一节提到，操作合同可以定义操作是否返回一个值，WSDualHttpBinding 允许进行双向通信。

这些都是消息模式，消息模式有 3 种类型：

- **请求/响应消息传输：**交换消息的“一般”方式，每个发送给服务的消息都会从客户机中得到一个返回的响应。这并不意味着客户机等待响应，因为可以用一般方式异步调用操作。
- **单向消息传输：**消息从客户机传输给 WCF 操作，但不发送响应。不需要响应时，就可以使用这种消息模式。例如，创建一个 WCF 操作，它会重启 WCF 宿主服务器，此时不需要等待响应。
- **双向消息传输：**一种较高级的模式，客户机可以用作服务器，服务器也可以用作客户机。启动后，双向消息传输允许客户机和服务器彼此发送消息，这些消息可能有响应，也可能没有。这类似于创建一个对象，并订阅该对象公开的事件。

本章的后面将使用这些消息模式。

## 26.2.5 行为

行为(behavior)是把没有直接提供给客户机的其他配置应用于服务和操作的方式。给服务添加行为，可以控制宿主进程如何实例化和使用行为，行为如何参与事务处理，在服务中如何解决多线程问题等。操作行为可以控制在操作执行过程中是否使用模仿功能，各个操作行为如何影响事务处理等。

在.NET 4 中，可以在不同的级别上指定默认行为，而无需给每个服务和操作指定每个行为的各个方面。还可以在需要时提供默认设置，重写设置，减少所需的配置量。

本章仅介绍 WCF 服务的基本知识，讨论行为的最基本功能。

## 26.2.6 驻留

本章的引言曾提到，WCF 服务可以存储在几个不同进程中，包括：

- **Web 服务器：**驻留在 IIS 的 WCF 服务是 WCF 提供的最接近 Web 服务的服务。还可以使用 WCF 服务中的高级功能和安全特性，这些功能和特性很难在 Web 服务中实现。也可以集成 IIS 特性，如 IIS 安全特性。
- **可执行文件：**可以把 WCF 服务驻留在.NET 中创建的任意应用程序类型中，如控制台应用程序、Windows 窗体应用程序和 WPF 应用程序。
- **Windows 服务：**可以把 WCF 服务驻留在 Windows 服务中，这表示可以使用 Windows 服务提供的有用特性，包括自动启动和错误恢复。
- **Windows Activation Service(WAS)：**专门用于驻留 WCF 服务，基本上是 IIS 的一个简化版本，可以在任何没有 IIS 的地方使用。

上述列表中的两个选项 IIS 和 WAS 为 WCF 服务提供了有用的特性，例如激活、处理循环和对象池。如果使用另外两个驻留选项，WCF 服务就是自驻留的。这未必是件坏事，因为我们可能不需要驻留环境提供的其他功能。但是自驻留的服务需要编写更多的代码。

## 26.3 WCF 编程

前面介绍了基础知识，下面开始编写一些代码。本节首先看一个在 Web 服务器上驻留的简单 WCF 服务和一个控制台客户程序。介绍了所创建的代码结构后，学习 WCF 服务和客户应用程序的基本结构。此后详细探讨一些重要主题：

- 定义 WCF 服务合同
- 自驻留的 WCF 服务

### 试一试：一个简单的 WCF 服务和客户程序

- (1) 在 C:\BegVCSharp\Chapter26 目录中创建一个新的 WCF 服务应用程序项目 Ch26Ex01。
- (2) 在解决方案中添加一个控制台应用程序 Ch26Ex01Client。
- (3) 在 Build 菜单上单击 Build Solution 选项。
- (4) 在 Solution Explorer 中右击 Ch26Ex01Client 项目，选择 Add Service Reference 选项。
- (5) 在 Add Service Reference 对话框中，单击 Discover。
- (6) 启动开发 Web 服务器，加载 WCF 服务的信息后，展开该引用，查看其细节，如图 26-2 所示(读者的端口号可能与本图中的不同)。

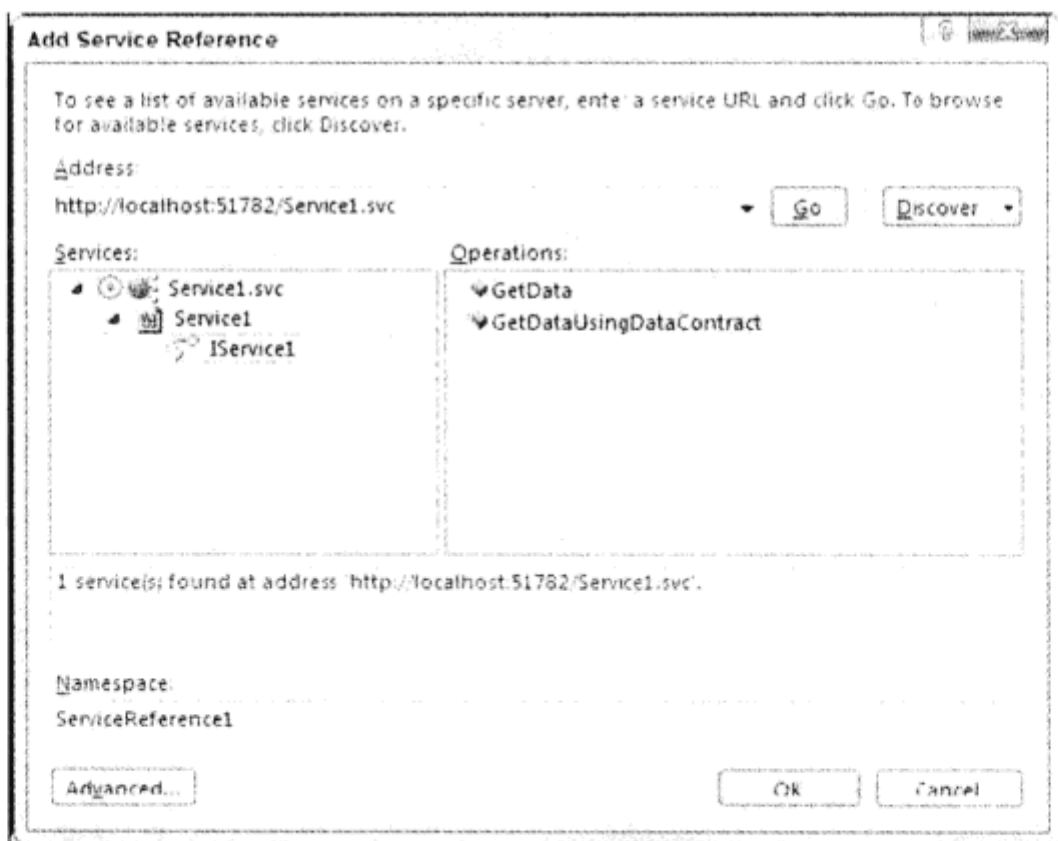


图 26-2

- (7) 单击 OK 按钮，添加服务引用。
- (8) 在 Ch26Ex01Client 应用程序中修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch26Ex01Client.ServiceReference1;

namespace Ch26Ex01Client
{
    class Program
    {
        static void Main(string[] args)
        {
            string numericInput = null;
            int intParam;
            do
```

```
{
    Console.WriteLine(
        "Enter an integer and press enter to call the WCF service.");
    numericInput = Console.ReadLine();
}
while (!int.TryParse(numericInput, out intParam));
Service1Client client = new Service1Client();
Console.WriteLine(client.GetData(intParam));
Console.WriteLine("Press an key to exit.");
Console.ReadKey();
}
}
```

代码段 Ch26Ex01Client\Program.cs

- (9) 在 Solution Explorer 中右击解决方案，选择 Set as StartUp Projects 选项。
- (10) 运行应用程序。出现提示后，单击 OK 按钮启用 Web.config 中的调试功能。在控制台应用程序窗口中输入一个数字，按下回车键。结果如图 26-3 所示。

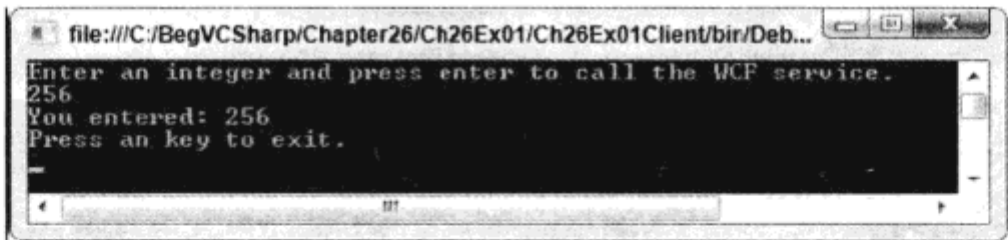


图 26-3

- (11) 退出应用程序，在 Solution Explorer 中右击 Ch26Ex01 项目中的 Service1.svc 文件，单击 View in Browser。
- (12) 查看窗口中的信息，如图 26-4 所示。

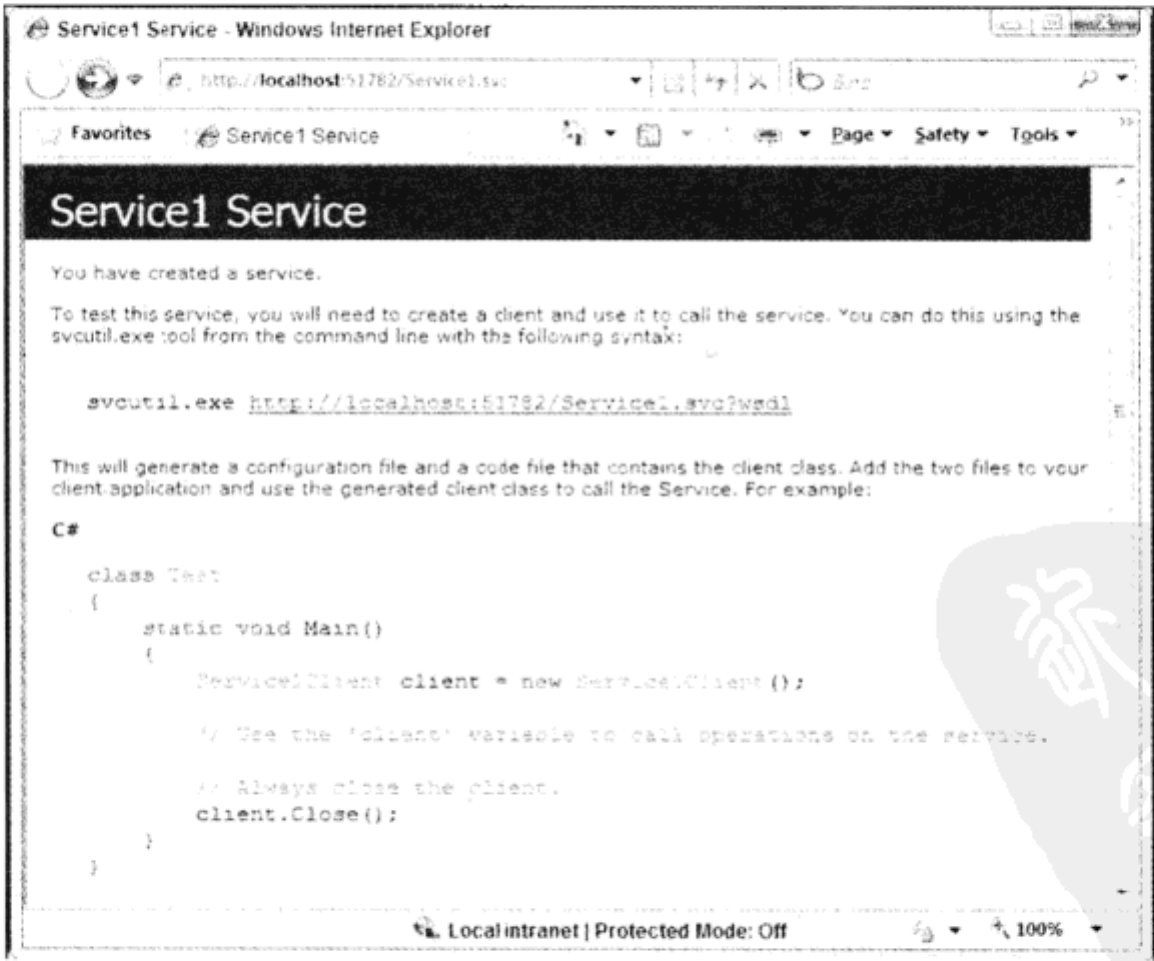


图 26-4



(13) 单击 Web 页面顶部的链接, 查看服务的 WSDL。现在还不需要了解 WSDL 文件中的所有内容的含义。

#### 示例的说明

这个示例中创建了一个驻留在 Web 服务器上的简单 Web 服务和控制台客户程序。我们为 WCF 服务项目使用了默认的 VS 模板, 这说明不必添加任何代码, 而使用这个默认模板中定义的一个操作 `GetData()`。对于这个示例, 使用什么操作并不重要, 而应关注代码的结构及其工作方式。

首先看看服务器项目 `Ch26Ex01`, 它包含:

- `Service1.svc` 文件, 它定义了服务的宿主。
- 类定义 `CompositeType`, 它定义了服务使用的数据合同(位于 `IService1.cs` 代码文件中)。
- 接口定义 `IService1`, 它定义了服务合同和两个操作合同。
- 类定义 `Service1`, 它实现 `IService1` 接口, 定义了服务的功能(位于 `Service1.svc.cs` 代码文件)。
- 配置段 `<system.serviceModel>`(在 `Web.config` 中), 它配置了服务。

`Service1.svc` 文件包含如下代码行。要查看这行代码, 应在 Solution Explorer 中右击该文件, 再单击 View Markup:



可从  
wrox.com  
下载源代码

```
< %@ ServiceHost Language="C#" Debug="true" Service="Ch26Ex01.Service1"
CodeBehind="Service1.svc.cs"%>
```

代码段 Ch26Ex01\Service1.svc

这是一个 `ServiceHost` 指令, 用于告诉 Web 服务器(本例是 Web 开发服务器, 尽管这也应用于 IIS)把什么服务存储在这个地址上。定义服务的类在 `Service` 属性中声明, 定义这个类的代码文件在 `CodeBehind` 属性中声明。这个指令是必须有的, 以获得 Web 服务器的驻留功能, 如前面几节所述。

显然, 没有驻留在 Web 服务器上的 WCF 服务不需要这个文件。本章后面将学习自驻留的 WCF 服务。

接着在 `IService1.cs` 文件中定义数据合同 `CompositeType`。从代码中可以看出, 数据合同只是一个类定义, 在类定义中包含了 `DataContract` 特性, 在类成员上包含了 `DataMember` 特性:



可从  
wrox.com  
下载源代码

```
[DataContract]
public class CompositeType
{
    bool boolValue = true;
    string stringValue = "Hello";

    [DataMember]
    public bool BoolValue
    {
        get { return boolValue; }
        set { boolValue = value; }
    }

    [DataMember]
    public string StringValue
    {
        get { return stringValue; }
        set { stringValue = value; }
    }
}
```



这个数据合同通过元数据提供给客户应用程序(查看示例中的 WSDL 文件, 就会看到这些元数据)。这允许客户应用程序定义一个类型, 该类型可以序列化到窗体上, 该窗体又可以由服务反序列化到 `CompositeType` 对象上。客户程序不需要知道这个类型的定义, 实际上, 客户程序使用的类可以有不同的实现代码。定义数据合同的这种方式虽简单但非常强大, 允许在 WCF 服务及其客户程序之间交换复杂的数据结构。

`IService1.cs` 文件还包含服务合同, 该服务合同定义为带有 `[ServiceContract]` 特性的接口。这个接口也在服务的元数据中进行了完整的描述, 并可以在客户应用程序中重建。接口成员构成了服务的操作, 每个操作都应用 `OperationContract` 特性创建一个操作合同。示例代码包含两个操作, 每个操作都使用了前面的数据合同:

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    string GetData(int value);

    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType composite);
}
```

前面介绍的 4 个合同定义属性都可以用特性进一步配置, 如下一节所述。实现服务的代码与其他类定义类似:



```
public class Service1 : IService1
{
    public string GetData(int value)
    {
        return string.Format("You entered: {0}", value);
    }

    public CompositeType GetDataUsingDataContract(CompositeType composite)
    {
        if (composite == null)
        {
            throw new ArgumentNullException("composite");
        }
        if (composite.BoolValue)
        {
            composite.StringValue += "Suffix";
        }
        return composite;
    }
}
```

注意这个类定义不需要继承自特定的类型,也不需要任何特定的特性,只需实现定义了服务合同的接口。实际上,可以在这个类及其成员中添加属性,以指定行为,但这些都**不是**强制的。

把服务的实现代码(类)和服务合同(接口)分开的效果极佳。客户程序不需要了解类的任何信息,类包含的功能可能远远超过了服务实现的功能。一个类甚至可以实现多个服务合同。

最后来分析 Web.config 文件中的配置。在配置文件中, WCF 服务的配置是从 .NET 远程技术中提取出来的一个特性,可以处理所有类型的 WCF 服务(非自驻留的服务和自驻留的服务)和 WCF 服务的客户程序(稍后介绍)。这个配置的词汇允许把任何配置应用于服务,甚至可以扩展其语法。

WCF 配置代码包含在 Web.config 或 app.config 文件的配置段 <system.serviceModel> 中。这个示例没有进行很多服务配置,因为使用了默认值。在 Web.config 文件中,配置段包含一个子段,它为服务行为 <behaviors> 重写了默认值。Web.config 中 <system.serviceModel> 配置段的代码如下(为了简洁起见,删除了注释):

可从  
wrox.com  
下载源代码

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

代码段 Ch26Ex01\Web.config

这个配置段可以在 <behavior> 子段中定义一个或多个行为,这些行为可以在多个其他元素上重用。可以给 <behavior> 段指定一个名称,以便进行重用(这样就可以在其他地方引用它),也可以不指定名称来使用(如本例所示),以指定重写默认的行为设置。



如果使用了非默认的配置,在 <system.serviceModel> 中就会包含一个 <services> 段,其中包含一个或多个 <service> 子段, <service> 段又可以包含 <endpoint> 子段,每个 <endpoint> 子段都定义了服务的一个端点。实际上,所定义的端点是服务的基端点。可以从中推断出操作的端点。

在 Web.config 中,重写的一个默认行为如下:

```
<serviceDebug includeExceptionDetailInFaults="false"/>
```

这个设置可以是 true,在传输给客户程序的任意错误中提供异常详情,通常只允许在开发过程中传输这些异常信息。

在 Web.config 中,另一个默认的重写行为与元数据相关。元数据允许客户程序获得 WCF 服务的描述。默认配置给服务定义了两个默认端点,一个端点由客户程序用于访问服务,另一个端点用于获得服务的元数据。在 Web.config 文件中,可以禁用这个功能,如下所示:

```
<serviceMetadata httpGetEnabled="false"/>
```

另外，还可以完全删除这行配置代码，因为默认行为不允许交换元数据。

如果在本例中尝试禁用这个功能，并不能阻止客户程序访问服务，因为客户程序已经在添加服务引用时获得了需要的元数据。但禁用元数据会禁止其他客户程序使用 Add Service Reference 工具访问这个服务。一般情况下，产品环境中的 Web 服务不需要提供元数据，所以应在开发阶段完成后禁用这个功能。

如果没有元数据，访问 Web 服务的另一种常见方式是在一个独立的程序集中为 WCF 服务定义合同，由宿主项目和客户项目引用。接着客户程序就可以直接使用这些合同生成一个代理，而不是通过元数据来访问服务。

前面介绍了 WCF 服务的代码，现在看看客户程序，尤其是使用 Add Service Reference 工具做了什么。注意在 Solution Explorer 中，客户程序包含一个文件夹 Service References，如果展开该文件夹，会看到一项 ServiceReference1，它是添加引用时选择使用的名称。

Add Service Reference 工具创建了访问服务所需的所有类。这包括服务的代理类，服务的代理类包含服务的所有操作方法(Service1Client)，以及从数据合同中生成的客户端类(CompositeType)。



也可以浏览 Add Service Reference 工具生成的代码(显示项目中的所有文件，包括隐藏的文件)，但目前最好不要浏览代码，因为有许多容易引起混淆的代码。

该工具还为项目添加了一个配置文件 app.config，这个配置定义了两个内容：

- 服务端点的绑定信息
- 端点的地址和合同

从服务描述中提取绑定信息，在客户程序中，每个可配置的选项都复制到配置文件中：



可从  
wrox.com  
下载源代码

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IService1"
          closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard"
          maxBufferSize="65536" maxBufferPoolSize="524288"
          maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8"
          transferMode="Buffered" useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None"
              proxyCredentialType="None" realm="" />
            <message clientCredentialType="UserName"
              algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

```

        </security>
    </binding>
</basicHttpBinding>
</bindings>

```

代码段 Ch26Ex01Client\app.config

这个绑定、服务的基地址(这是 Web 服务器存储的服务的.svc 文件地址)和合同的客户端版本 IService1 在端点配置中使用:

```

<client>
  <endpoint address="http://localhost:51782/Service1.svc"
    binding="basicHttpBinding"
    bindingConfiguration="BasicHttpBinding IService1"
    contract="ServiceReference1.IService1"
    name="BasicHttpBinding IService1" />
</client>
</system.serviceModel>
</configuration>

```

Add Service Reference 工具是非常全面的。实际上,大多数信息都不是必要的,可以用下面的代码替代这个配置文件:

```

<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="http://localhost:51782/Service1.svc"
        binding="basicHttpBinding"
        contract="ServiceReference1.IService1"
        name="BasicHttpBinding IService1" />
    </client>
  </system.serviceModel>
</configuration>

```

这段代码删除了整个<bindings>段和<endpoint>元素的 bindingConfiguration 属性,这表示客户端程序将使用默认的绑定配置。

但是为了学习 WCF 服务,掌握工具的全面性是非常重要的。它会显示包含在 BasicHttpBinding 默认绑定中的所有设置。本章不深入探讨 WCF 服务配置,但介绍了其中的一些配置,如超时设置,这些配置的命名很浅显,很容易理解。

这个示例介绍了许多基础知识,下面总结一下前面的内容:

- WCF 定义
  - 服务由服务合同接口定义,其中包括操作合同成员
  - 服务在实现了服务合同接口的类中实现
  - 数据合同只是使用数据合同特性的类型定义
- WCF 服务配置
  - 可以使用配置文件(Web.config 或 app.config)来配置 WCF 服务

- WCF Web 服务器驻留：
    - Web 服务器驻留把.svc 文件用作服务基地址
  - WCF 客户机配置：
    - 可以使用配置文件(web.config 或 app.config)来配置 WCF 服务客户机
- 下一节详细介绍合同。

26.3.1 WCF 测试客户程序

上面的示例创建了服务和客户程序，说明了 WCF 基本体系结构的工作原理，以及 WCF 服务的配置方式。但实际要使用的客户应用程序会比较复杂，也难以正确地测试服务。

为了便于开发 WCF 服务，VS 提供了一个测试工具，可用于确保 WCF 操作正常工作。这个工具会自动配置为处理 WCF 服务项目，所以如果运行项目，该工具就会显示出来。只需确保要测试的服务(即.svc 文件)设置为 WCF 服务项目的启动页面即可。另外，也可以把测试客户程序运行作为独立的应用程序运行。在 64 位操作系统上，测试客户程序位于 C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\WcfTestClient.exe。

如果使用 32 位操作系统，该路径是相同的，只是根文件夹是 Program Files。  
可以使用该工具调用服务操作，还可以用其他方式检查服务。如下面的示例所示。

试一试：使用 WCF 测试客户程序

- (1) 打开上一个示例中的 WCF Service Application 项目 Ch26Ex01。
- (2) 在 Solution Explorer 中右击 Service1.svc，选择 Set As Start Page。
- (3) 在 Solution Explorer 中右击 Ch26Ex01 项目，选择 Set As StartUp Project。
- (4) 在 Web.config 中，确保启用元数据。



可从  
wrox.com  
下载源代码

```
<serviceMetadata httpGetEnabled="true"/>
```

代码段 Ch26Ex01\Web.config

- (5) 运行应用程序。WCF 测试客户程序就会显示出来，如图 26-5 所示(需要一定的时间添加服务)。

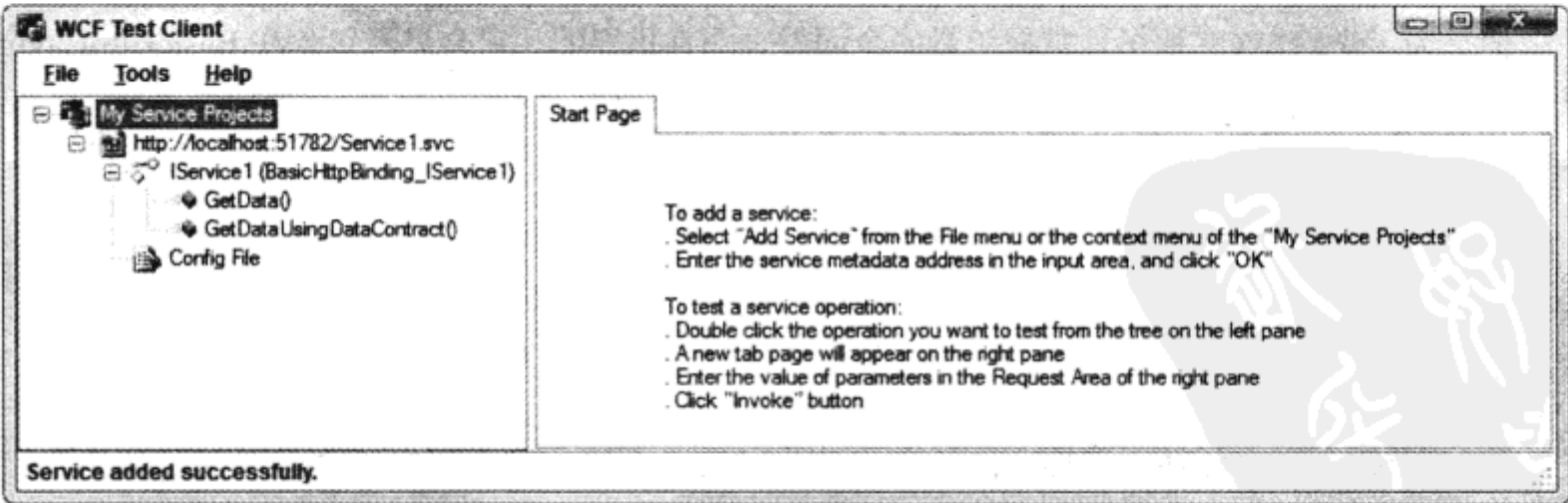


图 26-5

(6) 在测试客户程序的左面板上，双击 Config File。用于访问服务的配置文件就会显示在右面板上，如图 26-6 所示。

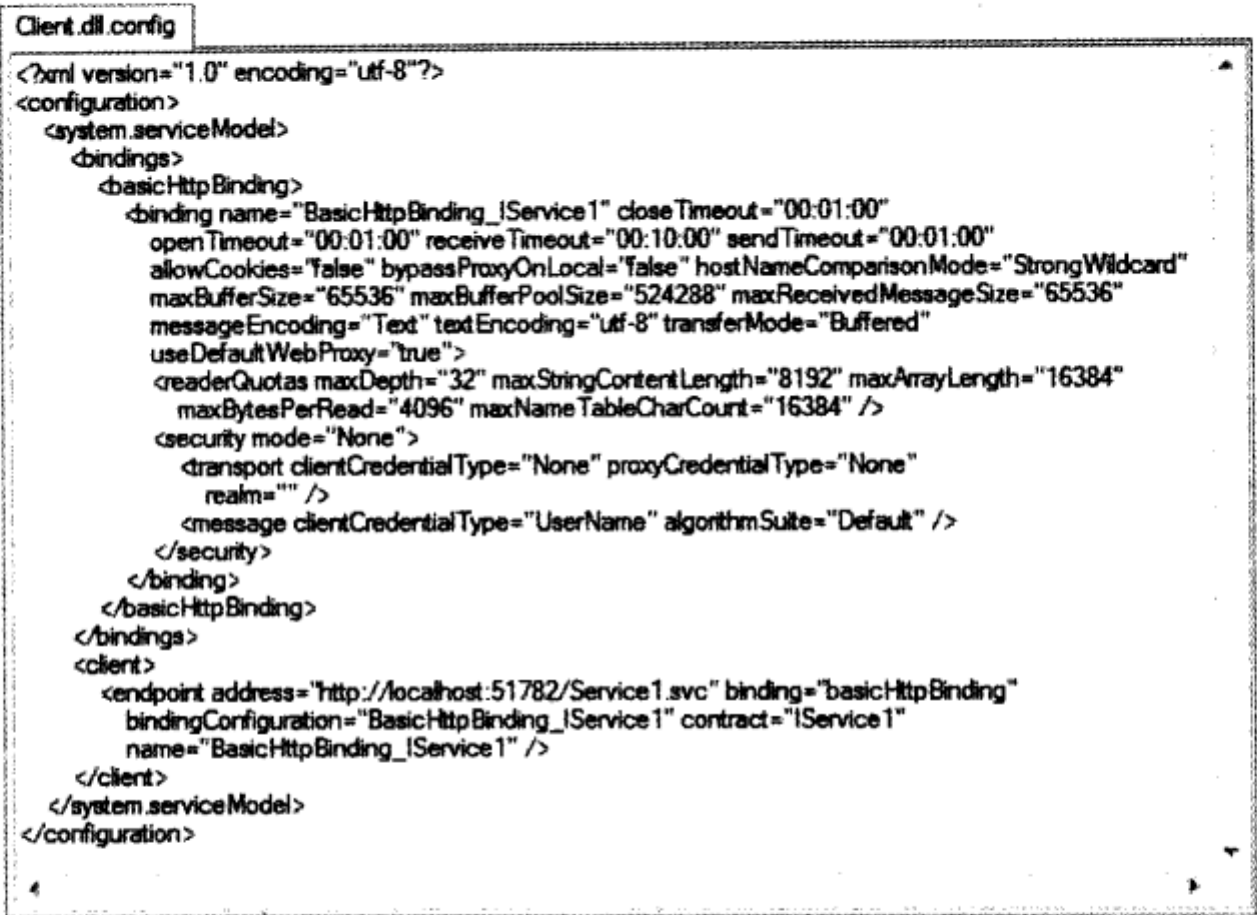


图 26-6

- (7) 在左面板上，双击 GetDataUsingDataContract()操作。
- (8) 在右面板上，把 BoolValue 的值改为 True，StringValue 改为 Test String，再单击 Invoke。
- (9) 如果显示了安全提示对话框，单击 OK 确认要把信息发送给服务。
- (10) 显示操作的结果，如图 26-7 所示。

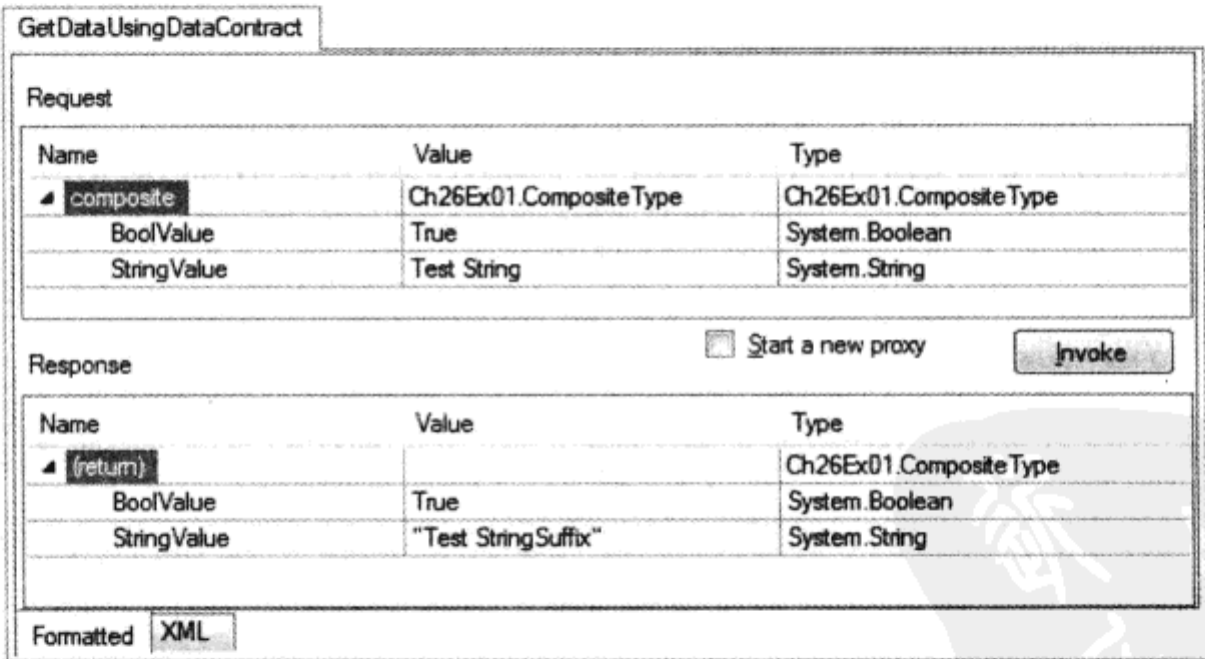


图 26-7

- (11) 单击 XML 标签页，查看请求和响应的 XML，如图 26-8 所示。



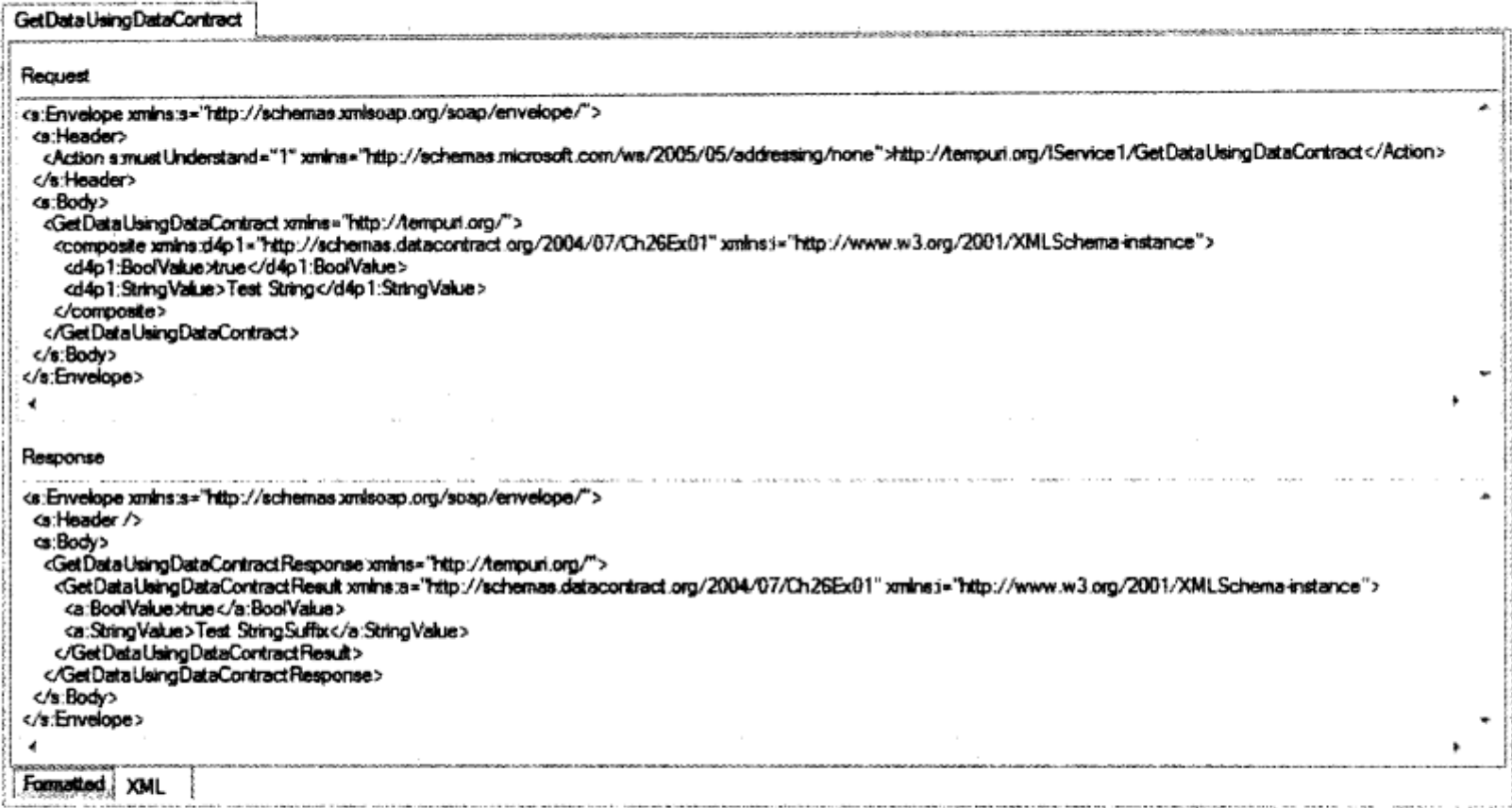


图 26-8

示例的说明

这个示例使用 WCF 测试客户程序在上一个示例创建的服务上检查和调用操作。首先注意，服务的加载有一点儿延迟，这是因为测试客户程序必须检查服务，以确定其功能。这个检查过程使用与 Add Service Reference 工具相同的元数据，所以必须确保元数据是可用的(在上一个示例中可能禁用了元数据)。检查完后，在工具的左面板上就会显示服务及其操作。

接着，查看用于访问服务的配置。与上一个示例中的客户应用程序一样，这些配置也是从服务的元数据中自动生成的，且包含在与服务相同的代码中。如有必要，可以通过该工具编辑这个配置文件，方法是右击 Config File 项，单击 Edit with SvcConfigeditor。

接着调用了一个操作。测试客户程序允许输入要使用的参数，并调用方法，然后显示结果，所有这些都不需要编写任何客户代码。我们还查看了为获得结果而发送和接收的 XML，这些信息的技术性很强，但在调试比较复杂的服务时，这些信息是绝对必需的。

26.3.2 定义 WCF 服务合同

从前面的示例可以了解到，通过 WCF 体系结构，可以结合使用类、接口和特性来方便地为 WCF 服务定义合同。本节将深入介绍这种技术。

1. 数据合同

要给服务定义数据合同，需要把 DataContractAttribute 特性应用于类定义。这个特性在名称空间 System.Runtime.Serialization 中。可以使用表 26-3 中所示的属性配置它。



表 26-3

属 性	说 明
Name	用不同于类定义的名称来命名数据合同。这个名称在 SOAP 消息和服务元数据定义的客户端数据对象上使用
Namespace	指定数据合同在 SOAP 消息中使用的名称空间

当需要与已有的 SOAP 消息格式交互操作时(类似于其他合同的指定属性), 需要使用这两个属性, 否则就不需要它们。

数据合同中的每个类成员都必须使用 `DataMemberAttribute` 特性, 它在名称空间 `System.Runtime.Serialization` 中。这个特性具有表 26-4 中所示的属性。

表 26-4

属 性	说 明
Name	指定序列化时数据成员的名称(默认为成员名称)
IsRequired	指定成员是否必须显示在 SOAP 消息中
Order	int 值, 指定序列化或反序列化成员的顺序, 如果一个成员必须在另一个成员之前出现, 这个顺序就是必须的。先处理 Order 较低的成员
EmitDefaultValue	把它设置为 false, 如果成员的值是默认值, 就禁止该成员包含在 SOAP 消息中

2. 服务合同

把 `System.ServiceModel.ServiceContractAttribute` 特性应用于接口定义, 就定义了服务合同。表 26-5 中所示的属性可用于定制服务合同。

表 26-5

属 性	说 明
Name	按照 WSDL 中<portType>元素中的定义, 指定服务合同的名称
Namespace	定义 WSDL 中<portType>元素使用的服务合同的名称空间
ConfigurationName	在配置文件中使用的服务合同名称
HasProtectionLevel	指定服务使用的消息是否有明确定义的保护级别。保护级别允许签名消息, 或者签名和加密消息
ProtectionLevel	保护级别, 用于保护消息
SessionMode	确定是否为消息启用会话。如果使用会话, 就可以确保关联上发送给服务的不同端点的消息, 即它们使用同一个服务实例, 因此可以共享状态
CallbackContract	对于双向消息传输, 客户机提供了合同和服务。这是因为, 如前所述, 双向通信中的客户机也用作服务器。这个属性允许指定客户机使用的合同

3. 操作合同

在定义服务合同的接口中，应用 `System.ServiceModel.OperationContractAttribute` 特性，就可以把成员定义为操作。这个特性具有表 26-6 中所示的属性。

表 26-6

属 性	说 明
Name	指定服务操作的名称。默认为成员名称
IsOneWay	指定操作是否返回一个响应。如果把它设置为 <code>true</code> ，则客户机不等待操作完成，就会继续执行
AsyncPattern	设置为 <code>true</code> ，操作就会实现为两个方法： <code>Begin&lt;methodName&gt;</code> 和 <code>End&lt;methodName&gt;</code> ，这两个方法可用于异步调用操作
HasProtectionLevel	参见表 26-5
ProtectionLevel	参见表 26-5
IsInitiating	如果使用会话，这个属性就确定调用这个操作是否可以启动新会话
IsTerminating	如果使用会话，这个属性就确定调用这个操作是否会中断当前会话
Action	如果使用寻址功能(WCF 服务的一个高级功能)，操作就有一个关联的动作名称，通过这个属性可以指定该名称
ReplyAction	同上，但为操作的响应指定动作名称

4. 消息合同

前面的示例中没有使用消息合同规范。如果使用消息合同，就应定义一个表示消息的类，再给类应用 `MessageContractAttribute` 特性。接着给这个类的成员应用 `MessageBodyMemberAttribute`、`MessageHeaderAttribute` 或 `MessageHeaderArrayAttribute` 特性。所有这些特性都在 `System.ServiceModel` 名称空间中。如果要高度控制 WCF 服务使用的 SOAP 消息，就不要使用消息合同，所以这里不详细讨论它。

5. 误合同

如果客户应用程序可以使用特定的异常类型，如定制异常，就可以给可能生成该异常的操作应用 `System.ServiceModel.FaultContractAttribute` 特性。在使用 WCF 时一般不希望这么做。

试一试：WCF 合同

- (1) 创建一个新 WCF 服务应用程序项目 `Ch26Ex02`，将其保存在 `C:\BegVCSharp\Chapter26` 目录中。
- (2) 给解决方案添加一个类库项目 `Ch26Ex02Contracts`，删除 `Class1.cs` 文件。
- (3) 在 `Ch26Ex02Contracts` 项目中添加对 `System.Runtime.Serialization` 和 `System.ServiceModel.dll` 程序集的引用。

(4) 在 Ch26Ex02Contracts 项目中添加 Person 类, 修改 Person.cs 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
```

```
namespace Ch26Ex02Contracts
{
    [DataContract]
    public class Person
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public int Mark { get; set; }
    }
}
```

代码段 Ch26Ex02Contracts\Person.cs

(5) 在 Ch26Ex02Contracts 项目中添加 IAwardService 类, 修改 IAwardService.cs 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace Ch26Ex02Contracts
{
    [ServiceContract(SessionMode=SessionMode.Required)]
    public interface IAwardService
    {
        [OperationContract(IsOneWay=true, IsInitiating=true)]
        void SetPassMark(int passMark);

        [OperationContract]
        Person[] GetAwardedPeople(Person[] peopleToTest);
    }
}
```

代码段 Ch26Ex02Contracts\IAwardService.cs

- (6) 对于 Ch26Ex02 项目, 添加对 Ch26Ex02Contracts 项目的引用。
- (7) 删除 Ch26Ex02 项目中的 IService1.cs 和 Service1.svc。
- (8) 在 Ch26Ex02 中添加一个新的 WCF 服务 AwardService。
- (9) 删除 Ch26Ex02 项目中的 IAwardService.cs 文件。

(10) 修改 AwardService.svc.cs 文件中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using Ch26Ex02Contracts;

namespace Ch26Ex02
{
    public class AwardService : IAwardService
    {
        private int passMark;

        public void SetPassMark(int passMark)
        {
            this.passMark = passMark;
        }

        public Person[] GetAwardedPeople(Person[] peopleToTest)
        {
            List < Person > result = new List < Person > ();
            foreach (Person person in peopleToTest)
            {
                if (person.Mark > passMark)
                {
                    result.Add(person);
                }
            }
            return result.ToArray();
        }
    }
}
```

代码段 Ch26Ex02\AwardService.svc.cs

(11) 修改 Web.config 中的服务配置段，如下所示：

```
<system.serviceModel>
  <protocolMapping>
    <add scheme="http" binding="wsHttpBinding"/>
  </protocolMapping>
  ...
</system.serviceModel>
```

(12) 打开 Ch26Ex02 的项目属性。在 Web 部分，选择 Specific port，输入端口号 51425，如图 26-9 所示。

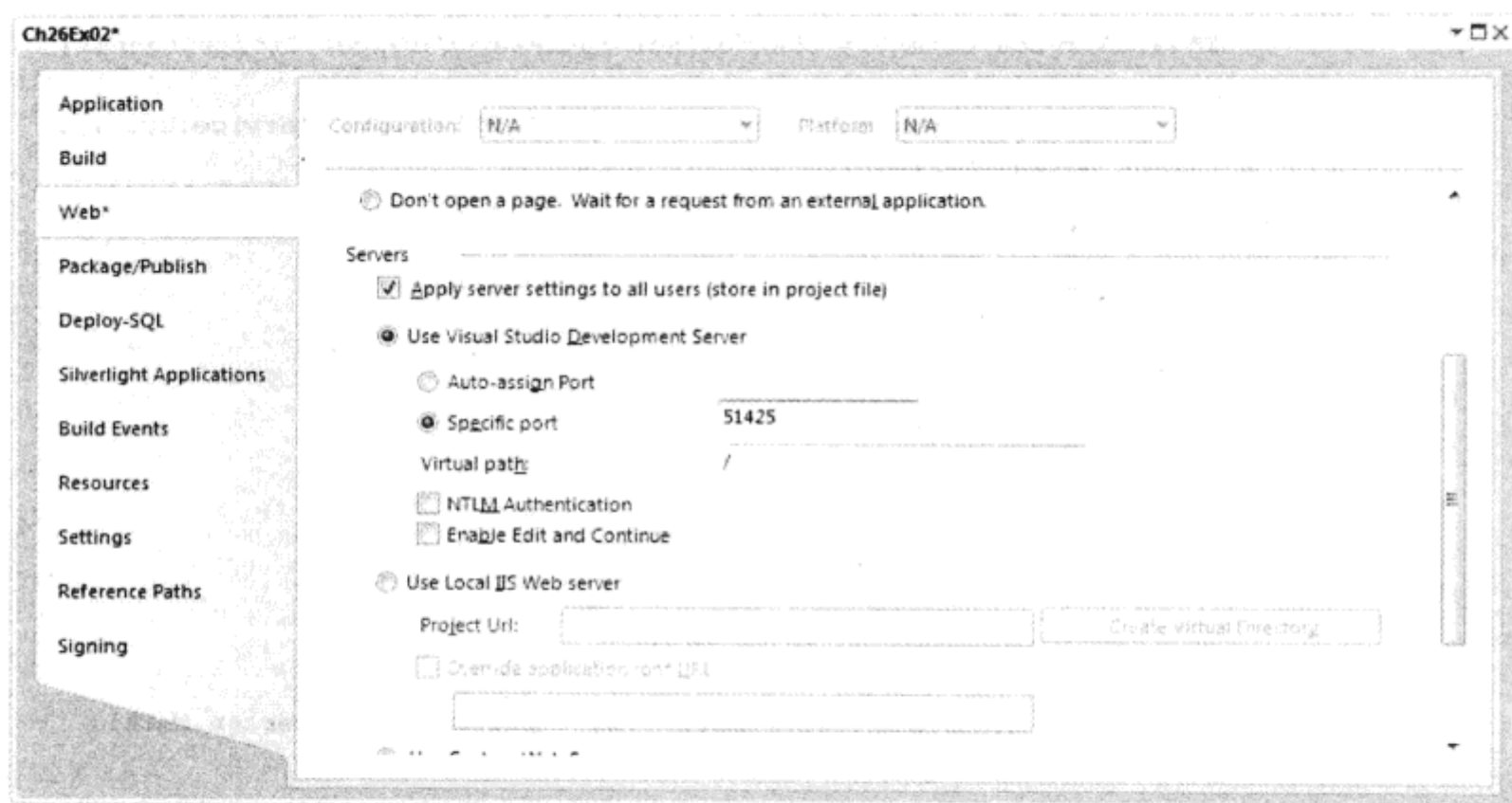


图 26-9

- (13) 在解决方案中添加一个新的控制台项目 Ch26Ex02Client，把它设置为启动项目。
- (14) 在 Ch26Ex02Client 项目中添加对 System.ServiceModel.dll 程序集和 Ch26Ex02Contracts 项目的引用。
- (15) 在 Ch26Ex02Client 项目中修改 Program.cs 中的代码，如下所示：



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using Ch26Ex02Contracts;

namespace Ch26E02Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Person[] people = new Person[]
            {
                new Person { Mark = 46, Name="Jim" },
                new Person { Mark = 73, Name="Mike" },
                new Person { Mark = 92, Name="Stefan" },
                new Person { Mark = 84, Name="George" },
                new Person { Mark = 24, Name="Arthur" },
                new Person { Mark = 58, Name="Nigel" }
            };

            Console.WriteLine("People:");
            OutputPeople(people);
        }
    }
}
```

```

        IAwardService client = ChannelFactory<IAwardService>.CreateChannel(
            new WSHttpBinding(),
            new EndpointAddress("http://localhost:51425/AwardService.svc"));
        client.SetPassMark(70);
        Person[] awardedPeople = client.GetAwardedPeople(people);

        Console.WriteLine();
        Console.WriteLine("Awarded people:");
        OutputPeople(awardedPeople);

        Console.ReadKey();
    }

    static void OutputPeople(Person[] people)
    {
        foreach (Person person in people)
        {
            Console.WriteLine("{0}, mark: {1}", person.Name, person.Mark);
        }
    }
}

```

代码段 Ch26Ex02Client\Program.cs

(16) 运行应用程序，结果如图 26-10 所示。

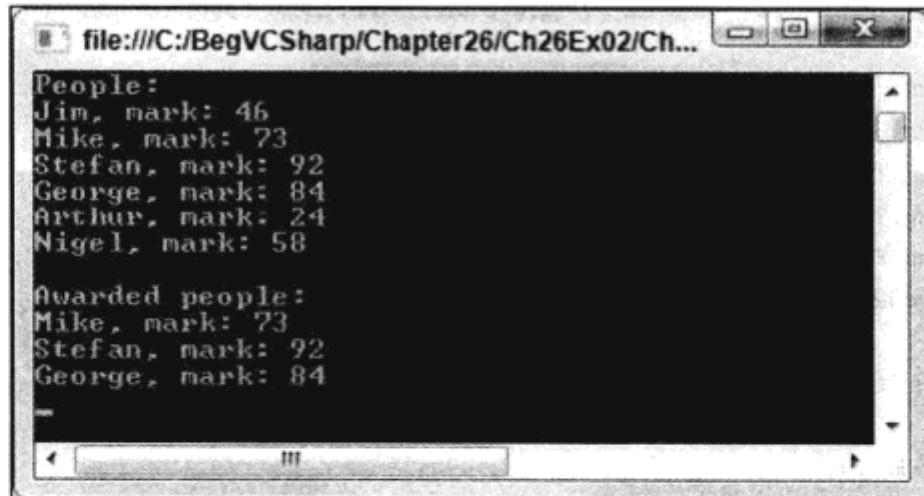


图 26-10

#### 示例的说明

这个示例在类库项目中创建了一系列合同，在 WCF 服务和客户程序中使用了这个类库。与前面的示例一样，这个服务也驻留在 Web 服务器上。这个服务的配置也被减少到最低程度。

在这个示例中，主要区别是客户程序不需要元数据，因为客户程序可以访问合同程序集。客户程序不是从元数据中生成一个代理类，而是通过另一种方法获得服务合同接口的引用。这个示例中另一个值得注意的地方是使用会话维护服务中的状态，这需要 WSHttpBinding 绑定，而不是 BasicHttpBinding 绑定。

这个示例使用的数据合同是一个简单的类 Person，它有一个 string 属性 Name 和一个 int 属性 Mark。使用的 DataContractAttribute 特性和 DataMemberAttribute 特性没有进行定制，也不需要给这

个合同重复迭代代码。

定义服务合同时，给 `IAwardService` 接口应用了 `ServiceContractAttribute` 特性。这个特性的 `SessionMode` 属性设置为 `SessionMode.Required`，因为这个服务需要状态：

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IAwardService
{
```

第一个操作合同 `SetPassMark()` 设置状态，因此其 `OperationContractAttribute` 的 `IsInitiating` 属性设置为 `true`。这个操作不返回任何值，所以将 `IsOneWay` 设置为 `true`，把操作定义为单向操作：

```
[OperationContract(IsOneWay=true, IsInitiating=true)]
void SetPassMark(int passMark);
```

另一个操作合同 `GetAwardedPeople()` 不需要进行任何定制，使用前面定义的数据合同：

```
[OperationContract]
Person[] GetAwardedPeople(Person[] peopleToTest);
}
```

这两个类型 `Person` 和 `IAwardService` 都可以用于服务和客户程序。服务在 `AwardService` 类型中实现了 `IAwardService` 合同，它不包含任何可标记的代码。这个类与前面的服务类的唯一区别是，这个类是有状态的。这是允许的，因为定义了一个会话，来关联来自客户程序的消息。

为了确保服务使用 `WSHttpBinding` 绑定，给服务添加了如下 `Web.config`：

```
<protocolMapping>
  <add scheme="http" binding="wsHttpBinding"/>
</protocolMapping>
```

这重写了 `HTTP` 绑定的默认映射。另外，也可以手工配置服务，保留已有的默认配置，但这个重写的配置要简单得多。注意这类重写配置会应用于项目中的所有服务。如果项目中有多个服务，就必须确保每个服务都能接受这个绑定。

客户程序比较有趣，主要是因为下面这行代码：

```
IAwardService client = ChannelFactory<IAwardService>.CreateChannel(
    new WSHttpBinding(),
    new EndpointAddress("http://localhost:51425/AwardService.svc"));
```

客户程序没有用 `app.config` 文件来配置与服务的通信，也没有从元数据中定义代理类，来与服务通信。而是通过 `ChannelFactory<T>.CreateChannel()` 方法创建代理类。这个方法创建了一个实现 `IAwardService` 客户程序的代理类，但在后台生成的类与服务通信，就像前面通过元数据生成的代理一样。



如果通过 `ChannelFactory<T>.CreateChannel()` 方法创建代理类，通信信道就默认为在 1 分钟后超时，导致通信错误。使连接一直处于激活状态有许多方式，但这些都超出了本章的讨论范围。

以这种方式创建代理类是一种非常有用的技术，可以快速生成客户应用程序。



### 26.3.3 自驻留的 WCF 服务

本章前面介绍了驻留在 Web 服务器上的 WCF 服务。它们可以在 Internet 上通信,但对于本地网络通信而言,这并不是最高效的方式。一方面,需要用计算机上的 Web 服务器驻留服务,另一方面,在应用程序的体系结构上出现一个独立的 WCF 服务可能并不合适。

因此应使用自驻留的 WCF 服务。自驻留的 WCF 服务存在于创建它的进程中,而不存在于特别建立的主机应用程序(如 Web 服务器)的进程中。这样,就可以使用控制台应用程序或 Windows 应用程序驻留服务了。

要建立自驻留的 WCF 服务,需要使用 `System.ServiceModel.ServiceHost` 类。用要驻留的服务类型或服务类的一个实例来实例化这个类。通过属性或方法可以配置服务宿主,也可以通过配置文件来配置。实际上,宿主进程(如 Web 服务器)使用 `ServiceHost` 实例完成该驻留任务。自驻留时,区别是直接与此类交互操作。但是,在宿主应用程序的 `app.config` 文件中, `<system.serviceModel>` 段中的配置使用的语法与本章前面的配置段中的相同。

可以通过任意协议提供自驻留的 WCF 服务,但是一般在这种类型的应用程序中使用 TCP 或指定管道绑定。通过 HTTP 访问的服务常常位于 Web 服务器进程中,因为可以获得 Web 服务器提供的额外功能,如安全性等。

如果要驻留 `MyService` 服务,可以使用下面的代码创建 `ServiceHost` 的一个实例:

```
ServiceHost host = new ServiceHost(typeof(MyService));
```

如果要存储 `MyService` 的实例 `MyServiceObject`, 可以编写如下代码, 创建 `ServiceHost` 的一个实例:

```
MyService myServiceObject = new MyService();  
ServiceHost host = new ServiceHost(myServiceObject);
```



**警告:** 只有配置了服务,使调用总是可以路由到同一个对象实例上,才能在 `ServiceHost` 中存储服务实例。为此,必须给服务类应用 `ServiceBehaviorAttribute` 特性,将这个特性的 `InstanceContextMode` 属性设置为 `InstanceContextMode.Single`。

创建了 `ServiceHost` 实例后,就可以通过属性配置服务、其端点和绑定。另外,如果把配置放在 `.config` 文件中,就会自动配置 `ServiceHost` 实例。

有了配置好的 `ServiceHost` 实例后,为了开始驻留服务,应使用 `ServiceHost.Open()` 方法。同样,通过 `ServiceHost.Close()` 方法可以停止驻留服务。第一次驻留 TCP 绑定的服务时,如果启用它,可能会收到 Windows 防火墙服务发出的一个警告,因为它阻塞了默认的 TCP 端口。只有给这个服务打开 TCP 端口,才能开始监听该端口。

下面的示例使用自驻留技术通过 WCF 服务提供 WPF 应用程序的一些功能。

#### 试一试: 自存储的 WCF 服务

- (1) 创建一个新的 WPF 应用程序 `Ch26Ex03`, 将其保存在 `C:\BegVCSharp\Chapter26` 目录中。
- (2) 使用 `Add New Item` 向导给项目添加一个新的 WCF 服务 `AppControlService`。
- (3) 修改 `MainWindow.xaml` 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch26Ex03.MainWindow"
  Title="Solar Evolution" Height="450" Width="430"
  Loaded="Window_Loaded" Closing="Window_Closing">
  <Grid Height="400" Width="400" HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Rectangle Fill="Black" RadiusX="20" RadiusY="20"
      StrokeThickness="10">
      <Rectangle.Stroke>
        <LinearGradientBrush EndPoint="0.358,0.02"
          StartPoint="0.642,0.98">
          <GradientStop Color="#FF121A5D" Offset="0"/>
          <GradientStop Color="#FFB1B9FF" Offset="1"/>
        </LinearGradientBrush>
      </Rectangle.Stroke>
    </Rectangle>
    <Ellipse Name="AnimatableEllipse" Stroke="{x:Null}" Height="0"
      Width="0" HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <Ellipse.Fill>
        <RadialGradientBrush>
          <GradientStop Color="#FFFFFFFF" Offset="0"/>
          <GradientStop Color="#FFFFFFFF" Offset="1"/>
        </RadialGradientBrush>
      </Ellipse.Fill>
      <Ellipse.Effect>
        <DropShadowEffect ShadowDepth="0" Color="#FFFFFFFF"
          BlurRadius="50"/>
      </Ellipse.Effect>
    </Ellipse>
  </Grid>
</Window>
```

代码段 Ch26Ex03\MainWindow.xaml

(4) 修改 MainWindow.xaml.cs 中的代码, 如下所示:



可从  
wrox.com  
下载源代码

```
using System.Windows.Shapes;
using System.ServiceModel;
using System.Windows.Media.Animation;

namespace Ch26Ex03
{
  /// <summary>
  /// Interaction logic for MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    private AppControlService service;
    private ServiceHost host;
    public MainWindow()
    {
```

```

        InitializeComponent();
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        service = new AppControlService(this);
        host = new ServiceHost(service);
        host.Open();
    }

    private void Window_Closing(object sender,
        System.ComponentModel.CancelEventArgs e)
    {
        host.Close();
    }

    internal void SetRadius(double radius, string foreTo,
        TimeSpan duration)
    {
        if (radius > 200)
        {
            radius = 200;
        }
        Color foreToColor = Colors.Red;
        try
        {
            foreToColor =
                (Color)ColorConverter.ConvertFromString(foreTo);
        }
        catch
        {
            // Ignore color conversion failure.
        }
        Duration animationLength = new Duration(duration);

        DoubleAnimation radiusAnimation = new DoubleAnimation(
            radius * 2, animationLength);
        ColorAnimation colorAnimation = new ColorAnimation(
            foreToColor, animationLength);
        AnimatableEllipse.BeginAnimation(Ellipse.HeightProperty,
            radiusAnimation);
        AnimatableEllipse.BeginAnimation(Ellipse.WidthProperty,
            radiusAnimation);
        ((RadialGradientBrush)AnimatableEllipse.Fill).GradientStops[1]
            .BeginAnimation(GradientStop.ColorProperty, colorAnimation);
    }
}
}

```

代码段 Ch26Ex03\MainWindow.xaml.cs

(5) 修改 IAppControlService.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
[ServiceContract]
public interface IAppControlService
{
    [OperationContract]
    void SetRadius(int radius, string foreTo, int seconds);
}
```

代码段 Ch26Ex03\IAppControlService.cs

(6) 修改 AppControlService.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
    private MainWindow hostApp;

    public AppControlService(MainWindow hostApp)
    {
        this.hostApp = hostApp;
    }

    public void SetRadius(int radius, string foreTo, int seconds)
    {
        hostApp.SetRadius(radius, foreTo, new TimeSpan(0, 0, seconds));
    }
}
```

代码段 Ch26Ex03\AppControlService.cs

(7) 修改 app.config 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Ch26Ex03.AppControlService">
        <endpoint address="net.tcp://localhost:8081/AppControlService"
          binding="netTcpBinding"
          contract="Ch26Ex03.IAppControlService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

代码段 Ch26Ex03\Web.config

(8) 在项目中添加一个新的控制台应用程序 Ch26Ex03Client。

(9) 在 Solution Explorer 中右击解决方案，单击 Set StartUp Projects。

(10) 配置解决方案，使之有多个启动项目，让多个项目同时启动，如图 26-11 所示。

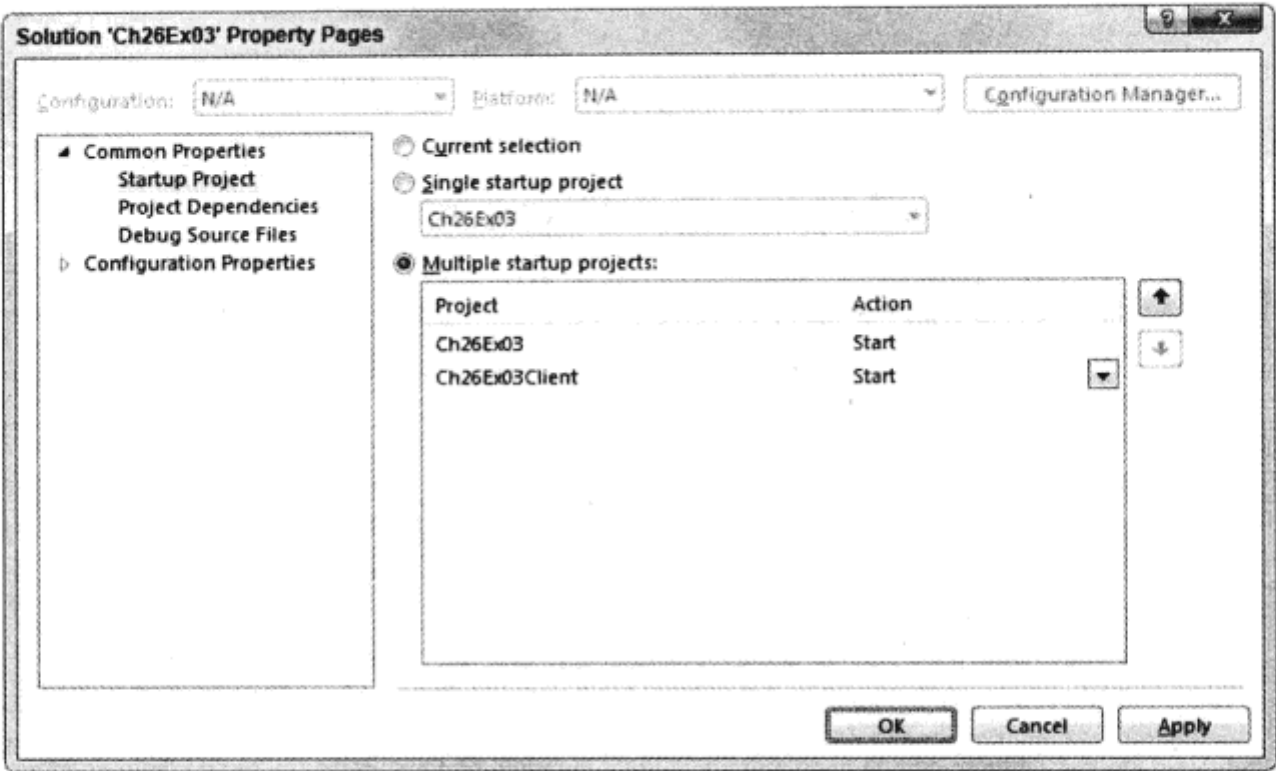


图 26-11

- (11) 在 Ch26Ex03Client 项目中添加对 System.ServiceModel.dll 和 Ch26Ex03 的引用。
- (12) 修改 Program.cs 中的代码，如下所示：



可从  
wrox.com  
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch26Ex03;
using System.ServiceModel;

namespace Ch26Ex03Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Press enter to begin.");
            Console.ReadLine();
            Console.WriteLine("Opening channel.");
            IAppControlService client =
                ChannelFactory<IAppControlService>.CreateChannel(
                    new NetTcpBinding(),
                    new EndpointAddress(
                        "net.tcp://localhost:8081/AppControlService"));
            Console.WriteLine("Creating sun.");
            client.SetRadius(100, "yellow", 3);
            Console.WriteLine("Press enter to continue.");
            Console.ReadLine();
            Console.WriteLine("Growing sun to red giant.");
            client.SetRadius(200, "Red", 5);
            Console.WriteLine("Press enter to continue.");
            Console.ReadLine();
            Console.WriteLine("Collapsing sun to neutron star.");
            client.SetRadius(50, "AliceBlue", 2);
        }
    }
}
```

```

        Console.WriteLine("Finished. Press enter to exit.");
        Console.ReadLine();
    }
}

```

代码段 Ch26Ex03Client\Program.cs

(13) 运行解决方案。出现提示时，打开 Windows 防火墙 TCP 端口，使 WCF 可以监听连接。

(14) 显示 Solar Evolution 窗口和控制台应用程序窗口时，在控制台窗口中按下回车键。结果如图 26-12 所示。

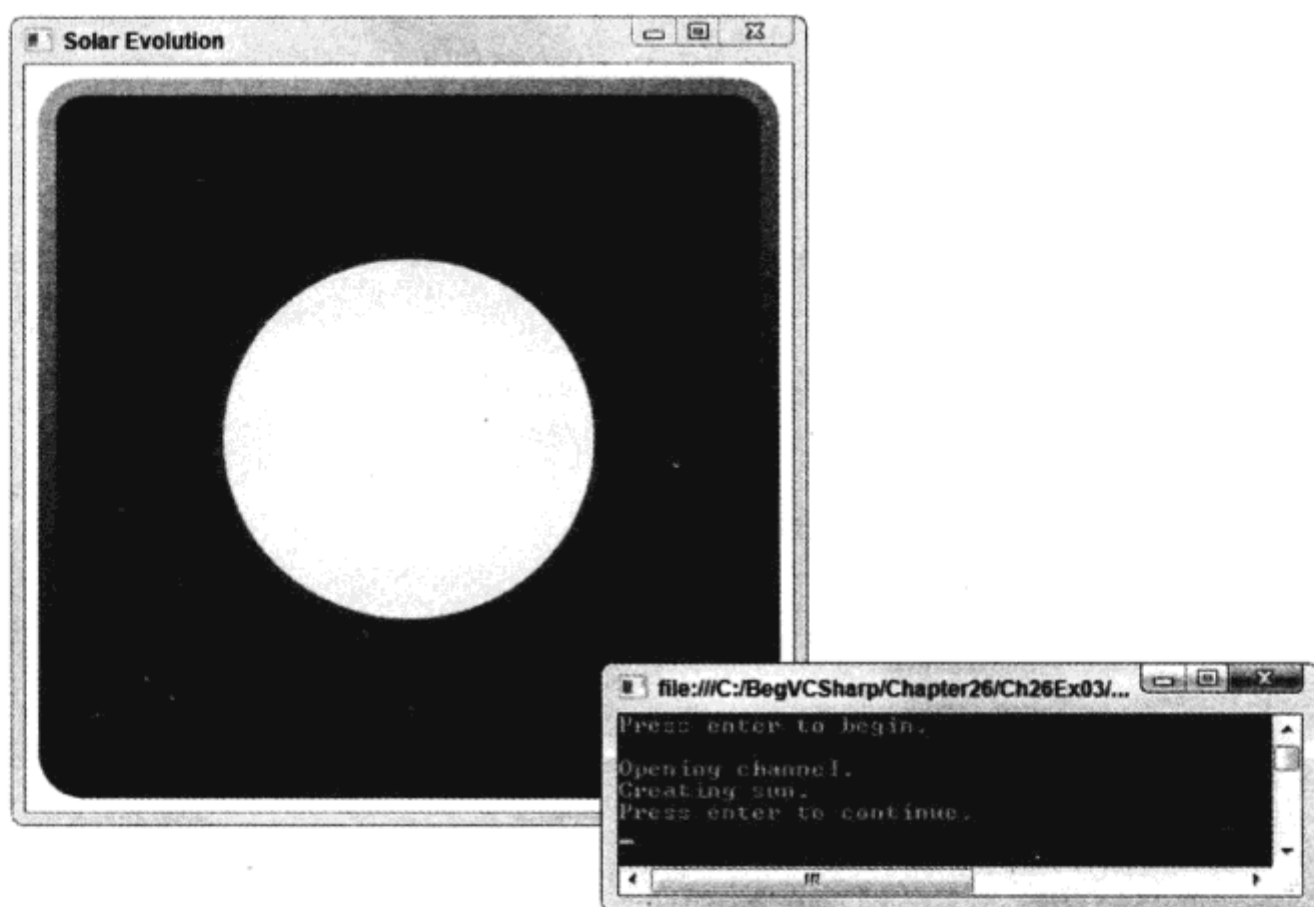


图 26-12

(15) 在控制台窗口中继续按下回车键，继续太阳演化循环。

#### 示例的说明

这个示例在 WPF 应用程序中添加了一个 WCF 服务，用它控制 Ellipse 控件的动画。我们创建了一个简单的客户应用程序来测试服务。如果不熟悉 WPF，不必过多地考虑示例中的 XAML 代码，我们只对 WCF 感兴趣。

WCF 服务 AppControlService 有一个操作 SetRadius()，客户程序调用这个操作来控制动画。这个方法与和它同名的方法通信，同名方法在 WPF 应用程序的 Window1 类中定义。为此，服务必须引用应用程序，所以必须寄宿该服务的一个对象实例。如前所述，这表示服务必须使用行为特性：

```

[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
    ...
}

```

在 Window1.xaml.cs 中, 在 Windows\_Loaded() 事件处理程序中创建服务实例。这个方法也为服务创建了一个 ServiceHost 对象, 并调用了其 Open() 方法, 以便开始寄宿:

```
public partial class MainWindow : Window
{
    private AppControlService service;
    private ServiceHost host;
    ...

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        service = new AppControlService(this);
        host = new ServiceHost(service);
        host.Open();
    }
}
```

在 Window\_Closing() 事件处理程序中, 应用程序关闭时, 寄宿过程中断。

配置文件非常简单, 它定义了 WCF 服务的一个端点, 监听端口 8081 的 net.tcp 地址, 使用默认的 NetTcpBinding 绑定:

```
<service name="Ch26Ex03.AppControlService">
  <endpoint address="net.tcp://localhost:8081/AppControlService"
    binding="netTcpBinding"
    contract="Ch26Ex03.IAppControlService"/>
</service>
```

这与客户应用程序中的代码相匹配:

```
IAppControlService client =
    ChannelFactory < IAppControlService > .CreateChannel(
        new NetTcpBinding(),
        new EndpointAddress(
            "net.tcp://localhost:8081/AppControlService"));
```

客户程序创建客户代理类时, 可以调用 SetRadius() 方法, 给它传递半径、颜色和动画持续时间等参数, 这些都会通过服务转发给 WPF 应用程序。接着, WPF 应用程序中的简单代码定义并使用动画, 来改变椭圆的大小和颜色。

如果使用的计算机名不是 localhost, 网络允许在指定的端口上通信, 这段代码就可以在网络上工作。另外, 还可以把客户程序和宿主应用程序进一步分离, 并通过 Internet 连接起来。无论采用什么方式, WCF 服务都提供了很好的通信方式, 建立它不需要付出过多努力。

## 26.4 小结

本章介绍了使用 WCF 服务在应用程序、进程和计算机之间通信的基本技术。首先学习了 WCF 服务的概念, 它与 Web 服务或远程实现的区别, 使用 WCF 服务需要了解的概念。接着讨论了如何编写 WCF 服务, 如何在客户程序中使用 WCF 服务, 如何以各种方式存储 WCF 服务。

这是在应用程序中使用 WCF 服务所必需的最少知识。本章仅触及 WCF 服务的皮毛, 尤其是 config 文件配置和行为。WCF 架构允许集成高级的安全体系, 以任意方式定制通信。



如果希望学习 WCF 服务的更多知识,可以阅读 *Professional WCF 4*(Wrox, 2010 年 6 月)。第 27 章将介绍 .NET 3.5 中的最后一种重要的新技术: Workflow Foundation(在 .NET 4 中做了很大的改进)。

## 26.5 练习

(1) 下述哪些应用程序可以寄宿 WCF 服务?

- a. Web 应用程序
- b. Windows 窗体应用程序
- c. Windows 服务
- d. COM+应用程序
- e. 控制台应用程序

(2) 如果要与 WCF 服务交换 MyClass 类型的参数,应实施什么类型的合同?需要什么特性?

(3) 如果把 WCF 服务寄宿在 Web 应用程序中,应对服务使用的基端点进行什么扩展?

(4) 在自寄宿 WCF 服务时,必须设置 ServiceHost 类的属性,调用它的方法,来配置服务。对吗?

(5) 提供服务合同 IMusicPlayer 的代码,它定义了 Play()、Stop()和 GetTrackInformation()操作。在合适的地方使用单向方法。还要为这个服务定义什么其他合同?

附录 A 给出了练习答案。

## 26.6 本章要点

主 题	重 要 概 念
WCF 基础	WCF 提供了创建远程服务并与之通信的框架,它合并了 Web 服务和远程体系结构的元素,并使用新技术来达到该目标
通信协议	可以通过几个协议与 WCF 服务通信,包括 HTTP 和 TCP。这表示可以使用客户应用程序的本地服务,也可以使用其他计算机或网络上的服务。为此,应通过对应于该协议的绑定及需要的特性,在特定的端点上访问该服务。可以通过行为控制这些特性,例如使用会话状态或提供元数据。.NET 4 包含许多默认的设置,非常便于定义简单的服务
通信负载	对来自 WCF 服务的响应的调用一般编码为 SOAP 消息。也可以使用普通的 HTTP 消息,如果需要,还可以从头开始定义自己的负载类型
寄宿	WCF 服务可以寄宿在 IIS 或 Windows 服务中,也可以是自寄宿的。使用 IIS 等宿主来存储 WCF 服务,就可以利用该宿主内置的功能,包括安全性和应用程序池。自寄宿比较灵活,但可能需要更多的配置和编码
合同	通过合同定义 WCF 服务和客户代码之间的接口。服务及其提供的操作是通过服务和操作合同定义的。数据类型用数据合同定义。可使用消息合同和契约合同来进一步定制通信
客户应用程序	客户应用程序与 WCF 服务通过代理类来通信。代理类为服务实现了服务合同接口,对这个接口的操作的所有调用都重定向到服务上。可以使用 Add Service Reference 工具生成代理,也可以通过信道工厂方法以编程方式创建代理。为了成功通信,必须配置客户程序,以匹配服务的配置



# 第 27 章

## Windows Workflow Foundation

### 本章内容:

---

- 工作流的含义和执行工作流的方法
- 活动的含义
- 如何创建定制活动
- 如何从活动中发送电子邮件

Windows Workflow Foundation (WF)最初出现在.NET 3.0 中,在.NET 3.5 中做了修订,另添加了一些功能,使之更容易与 Windows Communication Foundation (WCF)集成在一起。.NET 4 完全重写了 Workflow,虽然其核心概念没有变化,但实现方式却完全不同。本章将介绍 Windows Workflow Foundation 4 (WF4)。

工作流的一个简化定义是“一个活动集合”,但这不是一个令人完全满意的定义。使用一个类比方法可能更加有效。

编写程序时,我们会使用语句(例如 if/else),调用函数(Console.WriteLine),也会执行循环中的一些代码。但终端用户可能并不理解编程,所以他们会向我们反映,他们希望系统做什么工作,而我们编写代码,以便满足他们的要求。

现在假定可以为终端用户提供一个做了极大简化的编程环境,在这个环境中,预先建立了语句和流程控制逻辑,终端用户只需把这些部分插入他们希望的位置即可。此时就可以使用 Workflow 4。语句和控制逻辑都称为活动,这些活动可以插入到工作流中。

### 27.1 Hello World

每本编程图书都需要一个 Hello World 示例,本书也不例外。但这个示例不使用传统的编程语言,而使用 Workflow 4。下面的示例将创建一个 Workflow 项目,添加一个活动,并执行该工作流。

试一试：简单的 Workflow 4 应用程序

(1) 在 Visual Studio 2010 中，创建一个新的 Workflow Console Application 项目，确保在屏幕上部的下拉列表中选择 .NET Framework 4，如图 27-1 所示。

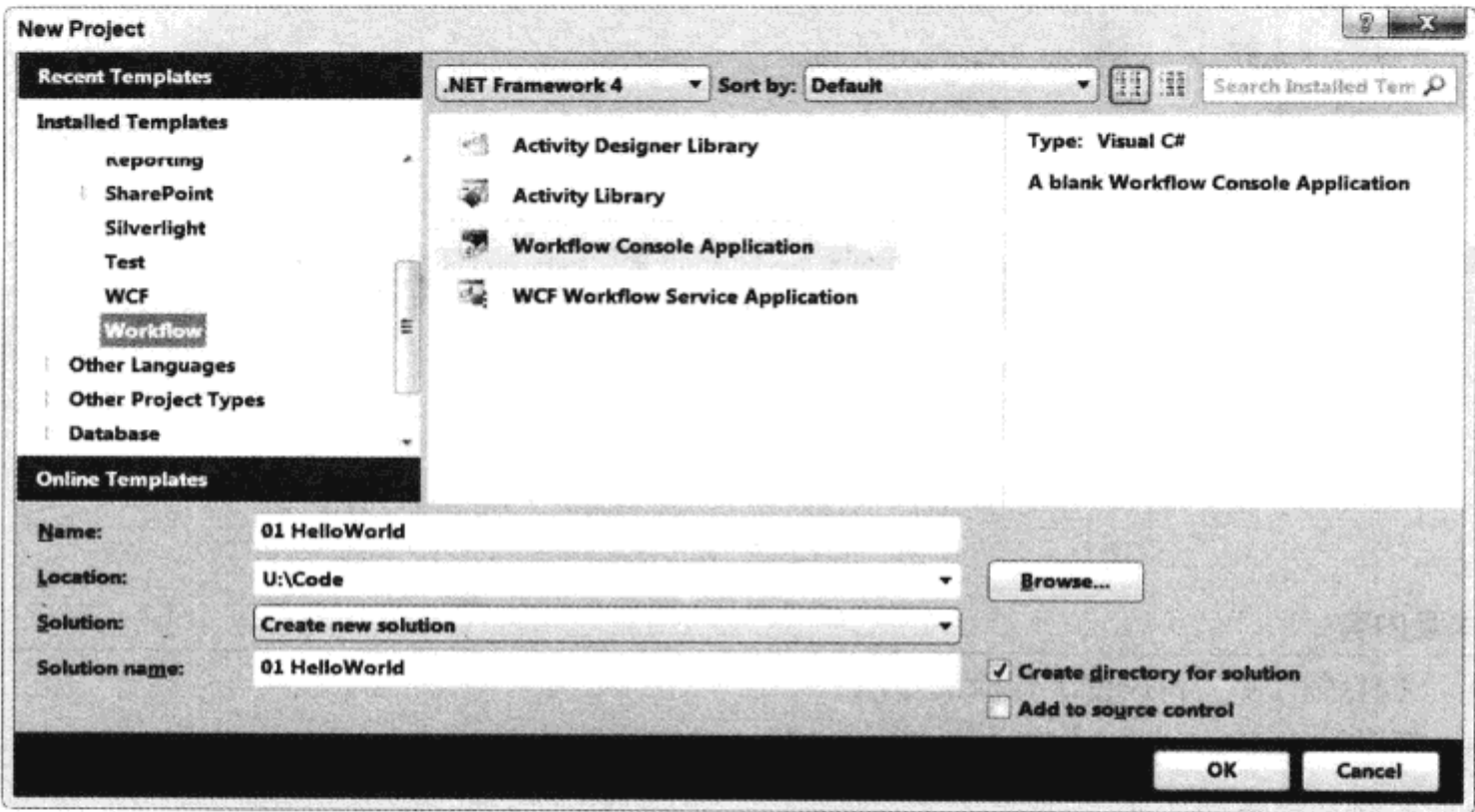


图 27-1

- (2) 从工具箱中把一个 WriteLine 活动拖放到主设计区中。
- (3) 在文本框中键入 Hello Workflow World，如图 27-2 所示。

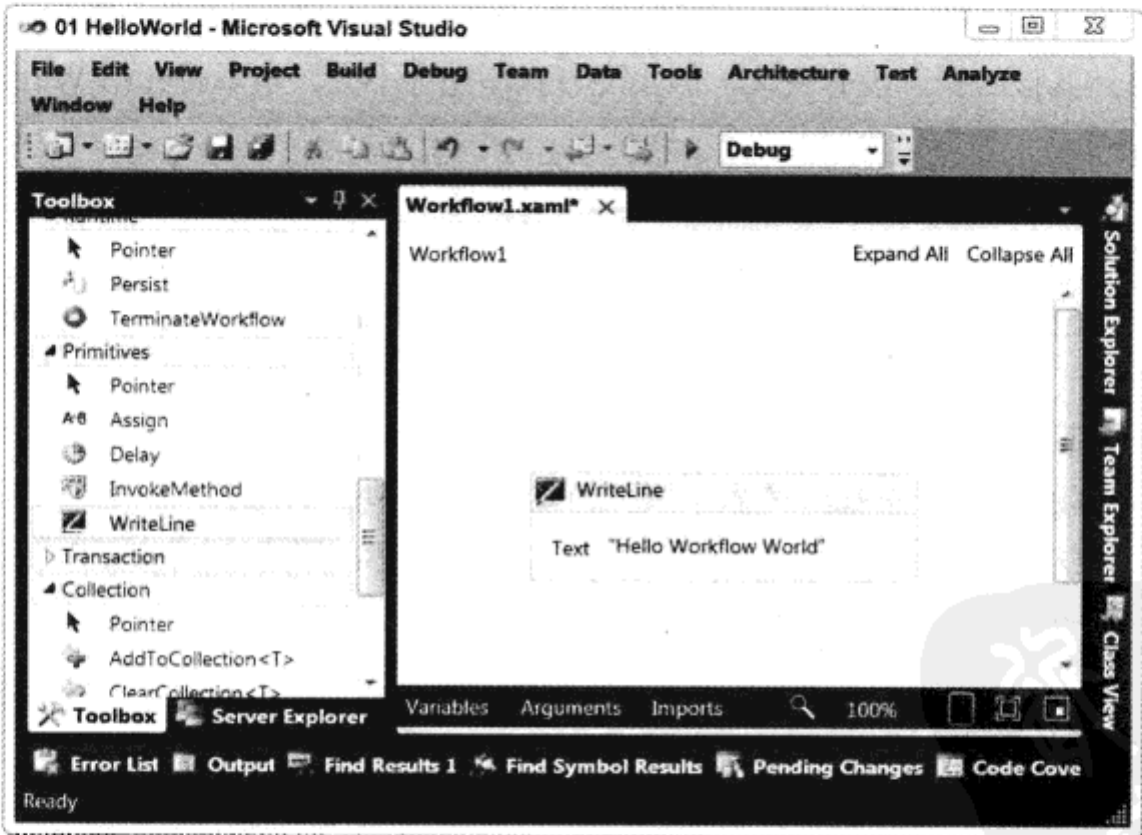


图 27-2

- (4) 运行应用程序，查看输出的文本。

示例的说明

运行应用程序时，会执行工作流中的活动。在这个示例中只有一个活动，它输出一些文本。活动执行完毕后，工作流就完成了，应用程序随之退出。当然，可以在工作流中提供更多活动，执行更有效的任务，而不是仅仅在控制台上输出一条消息而已。

27.2 工作流和活动

上一节列举了一个简短的工作流例子，该工作流使用了一个简单的活动。工作流是一个活动集合，它定义了这些活动的执行顺序。该示例使用了顺序工作流，这种工作流包含了多个按顺序执行的活动。

活动是一个工作单元，分为两种类型。第一种是上例中的简单活动，例如 WriteLine 活动。此类活动只执行一个任务。另一种是复合活动，这种活动有几个熟悉的例子，例如 While 活动，它包含其他子活动。

因此，工作流类似于程序——工作流中的简单活动类似于常规的编程语句，流程控制活动类似于流程控制语句，执行方式也类似于程序。

如果工作流类似于程序，那么，我们可以像编程那样创建自己的函数吗？也许我们需要一个发送电子邮件的函数，或者把数据写入审核踪迹的函数。此时可以使用定制活动——可以编写这些低级的功能，用户只需把它们拖放到工作流中即可。

Windows Workflow Foundation 4 提供了许多活动，下一节将讨论其中的一些活动，说明如何在工作流中使用它们。

27.2.1 If 活动

这个活动的工作原理类似于 C# 中的 if/else 语句。执行 If 活动时，将计算一个条件，接着根据该条件的计算结果确定工作流的执行路径。

使用 If 活动时，它会显示在工作流中，如图 27-3 所示。

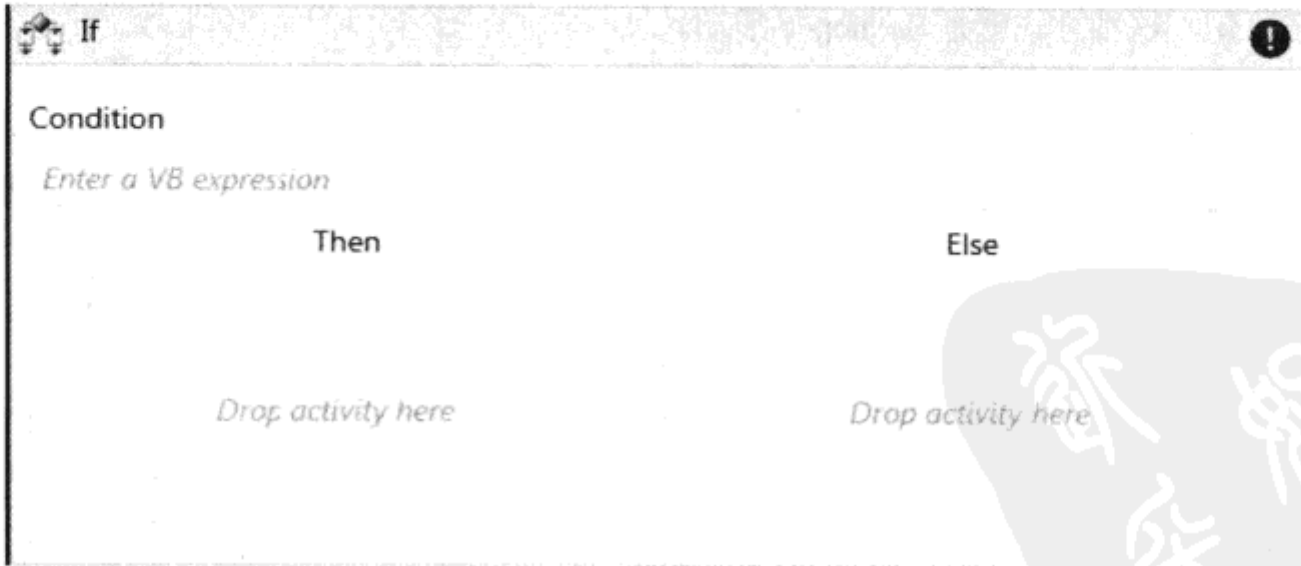


图 27-3

If 活动包含一个在执行活动时计算的条件表达式，以及用于 Then 和 Else 活动的占位符。Condition 属性是一个表达式，计算结果是一个布尔值，所以可以在这里包含任意有效的表达式。



Workflow 4 包含一个使用 Visual Basic 语法的表达式引擎，这对于 C#程序员而言是一个陌生的领域，因为 Visual Basic 与 C#大相径庭。但目前的情况就是这样，所以要使用内置的活动，必须学习 Visual Basic。只要记住 Visual Basic 的语法非常罗嗦，并且删除分号即可。

表达式可以引用工作流中定义的任意变量，也可以访问.NET Framework 中的许多静态类。所以可以根据 Environment.Is64BitOperatingSystem 值定义表达式(假定这对工作流的某部分非常重要)。一般情况下，可以定义传送到工作流中的变元，然后就可以在 If 活动中由表达式计算该变元。本章后面会讨论变元和变量。

27.2.2 While 活动

任何程序员都很熟悉 While 活动，它会计算一个条件，若计算结果为 true，就执行活动体，如图 27-4 所示。



图 27-4

While 体中只能包含一个活动，但大多数程序都要求循环中有多个语句，所以必须通过某种方式添加更多语句。这种方式就是 Sequence 活动。

27.2.3 Sequence 活动

这个活动允许构建一系列其他活动，在执行时，会从第一个子活动开始，然后依次执行每个子活动，如图 27-5 所示。

图 27-5 显示了一个 Sequence 活动，它包含 3 个子活动：一个 WriteLine、一个 While 和另一个 WriteLine。如果执行这个工作流，会把第一条消息写到控制台中，接着执行 while 循环，最后把另一条消息写到控制台中。图 27-5 还显示了工作流设计器的另一个有用特性：折叠(和展开)活动。本例子中折叠了 While 活动，只显示该活动的名字——活动右边带两个 V 形图标的按钮允许切换活动内容的可见性。

在设计大型工作流时，折叠和展开活动的功能非常有用，因为可以折叠当前暂不处理的工作流部分，展开正在处理的部分。

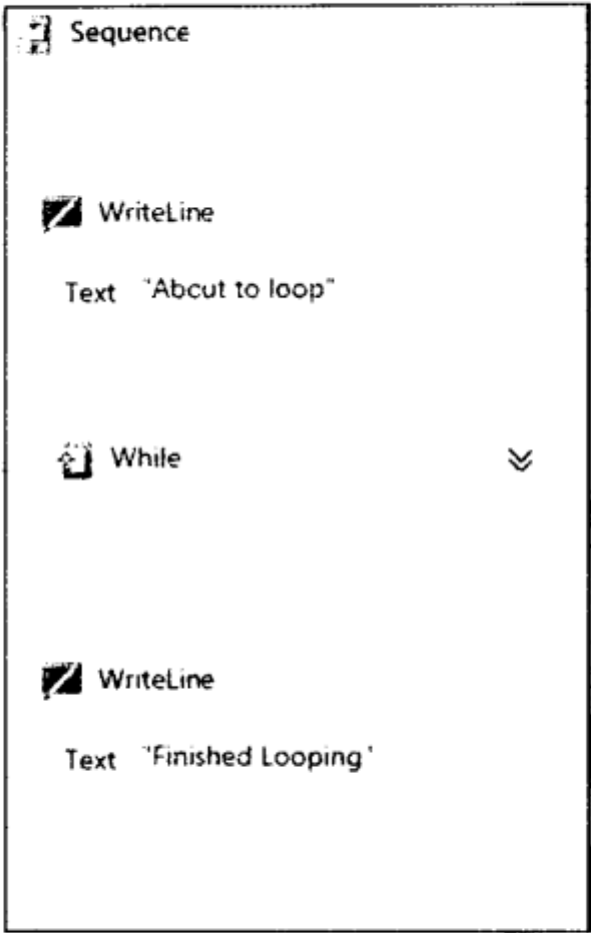


图 27-5

工作流设计器的其他特性使工作流变得更容易处理。例如，设计器右下角包含控件组，如图 27-6 所示。

这些控件(从左到右)可以把当前工作流放大到 100%，放大到某个定制的级别，使当前工作流的大小刚好能放在屏幕上，显示或隐藏概览窗口(显示工作流的缩略图)。另外，还可以双击某个复合活动，例如 While 或 If，以便隐藏所有更高级的活动，仅显示被单击的活动及其子活动。屏幕左上角的导览图显示了当前位置，如图 27-7 所示。



图 27-6

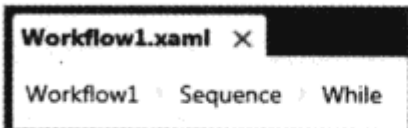


图 27-7

可以单击序列中的任意项，导航到工作流的更高层次，直至到达顶层为止。

### 27.3 变元和变量

在任何常规的编程语言中，都可以使用变元给函数传送值(以及检索响应)，可以在函数中定义变量，并把变量用作临时存储器。Workflow 也是一种编程语言，所以也可以使用这些结构。

但在进一步解释变元和变量之前，先考虑一下工作流的实际组成部分。如果在 Visual Studio 2010 中打开 Solution Explorer，查看本章第一部分创建的解决方案，会看到图 27-8 中突出显示的文件。



图 27-8



双击 Workflow1.xaml, 会在设计器中显示 workflow。但这只是一个普通的 XML 文件, 所以不要双击它, 而是右击, 再选择 Open With。在打开的对话框中选择 XML Editor, 就会打开相应文件, 显示组成 workflow 的 XML:

```
<Activity mc:Ignorable="sap"
  x:Class="_01_HelloWorld.Workflow1"
  mva:VisualBasic.Settings="Assembly references and imported
    namespaces serialized as XML namespaces"
  xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
  xmlns:mva="clr-namespace:Microsoft.
    VisualBasic.Activities;assembly=System.Activities"
  xmlns:s="clr-namespace:System;assembly=mscorlib"
  xmlns:s1="clr-namespace:System;assembly=System"
  xmlns:s2="clr-namespace:System;assembly=System.Xml"
  xmlns:s3="clr-namespace:System;assembly=System.Core"
  xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=System.Activities"
  xmlns:sap="http://schemas.microsoft.com/netfx/2009/xaml/activities/presentation"
  xmlns:scg="clr-namespace:System.Collections.Generic;assembly=System"
  xmlns:scg1="clr-namespace:System.
    Collections.Generic;assembly=System.ServiceModel"
  xmlns:scg2="clr-namespace:System.Collections.Generic;assembly=System.Core"
  xmlns:scg3="clr-namespace:System.Collections.Generic;assembly=mscorlib"
  xmlns:sd="clr-namespace:System.Data;assembly=System.Data"
  xmlns:sdl="clr-namespace:System.Data;assembly=System.Data.DataSetExtensions"
  xmlns:s1="clr-namespace:System.Linq;assembly=System.Core"
  xmlns:st="clr-namespace:System.Text;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WriteLine sad:XamlDebuggerXmlReader.FileName=
    "U:\Code\01 HelloWorld\01 HelloWorld\Workflow1.xaml"
    sap:VirtualizedContainerService.HintSize="209.6,200"
    Text="Hello Workflow World" />
</Activity>
```

代码中引用了许多 XML 名称空间, 但主要部分仅显示了一个带有 Text 属性的 WriteLine 活动。要创建传送给 workflow 的变元, 可以使用 workflow 设计器中的 Arguments 设计器。这个选项显示在设计器界面的左下角, 如图 27-9 所示。

Variables Arguments Imports

图 27-9

下面的示例将创建一个输入变元和一个变量, 并在一个简单的工作流中使用它们。

#### 试一试: 使用变元和变量

- (1) 在 Visual Studio 2010 中创建一个新的 Workflow Console Application 项目。
- (2) 显示 workflow 时, 单击 Arguments 按钮, 创建一个字符串变元, 如图 27-10 所示。把变元的名字设置为 Name, 方向设置为 In, 数据类型设置为 String。
- (3) 在工作流中添加一个 Sequence 活动, 再给 Sequence 添加一个 WriteLine 活动, 在表达式文

本框中键入 Name。注意不要包括引号，因为这是变元的名字，而不是字面量字符串。

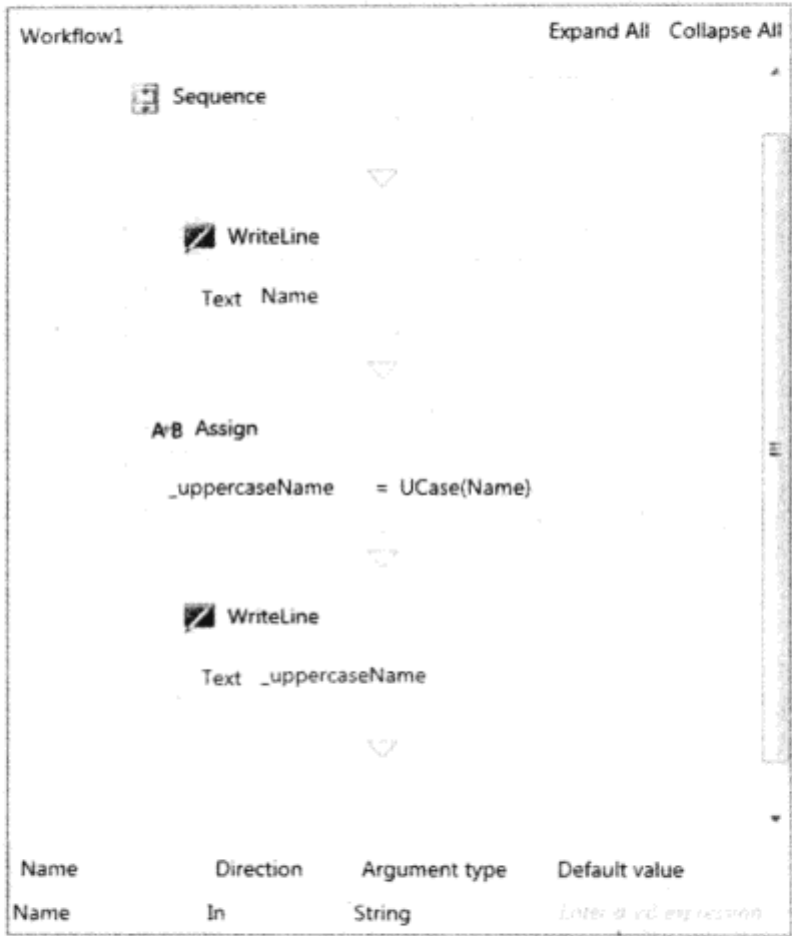


图 27-10

- (4) 现在单击 Variables 按钮，定义一个 String 类型的 \_uppercaseName 变量。这个变量将用于存储 Name 变元的大写值。
- (5) 把一个 Assign 活动拖放到设计器上。把该活动左边的文本框设置为 \_uppercaseName，右边的文本框设置为 UCase(Name)。这个活动给变量赋值，在本例中，该值是 VB 中的一个函数调用，该函数把其参数转换为大写字符串。
- (6) 把一个 WriteLine 活动拖放到设计器上，将其文本框设置为 \_uppercaseName。
- (7) 现在切换到 MainProgram.cs 文件，在其中给 Name 变元创建一个值，并传送给工作流。在该文件中包含如下代码：

```
class Program
{
    static void Main(string[] args)
    {
        WorkflowInvoker.Invoke(new Workflow1());
    }
}
```

- (8) 需要修改上面的代码，以传送 Name 变元的值，进行如下修改：

```
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, object> parms = new Dictionary<string, object>();
        parms.Add("Name", "Morgan");
        WorkflowInvoker.Invoke(new Workflow1(), parms);
    }
}
```

```
    }
}
```

如果愿意，还可以替换为自己的名字。

(9) 建立并运行项目，结果应是两行，其中一行是大写。

#### 示例的说明

执行这个应用程序时，会把一个字符串参数传送到工作流中。输入参数被数据绑定到 WriteLine 活动和 Assign 活动上，在 Assign 活动中用一个表达式把输入参数转换为大写形式，然后保存到一个本地变量中。这个变量供第二个 WriteLine 活动使用。

如果现在使用 XAML 编辑器打开示例中的 Workflow1.xaml 文件，就可以在该文件的顶部看到变元的定义：

```
<x:Members>
  <x:Property Name="Name" Type="InArgument(x:String)" />
</x:Members>
```

另外还可以看到序列中变量的定义：

```
<Sequence.Variables>
  <Variable x:TypeArguments="x:String" Name="_uppercaseName" />
</Sequence.Variables>
```

与常规程序中的变量一样，工作流中的变量也有作用域，定义了变量在何时是可用的。一旦定义变量的活动结束了，就会释放该变量。

在上面的代码示例中，创建了一个名称/值对字典，接着使用键 Name 把一个值添加到字典中。这里使用的键值必须准确匹配变元的定义，否则就不会正确设置变元值，工作流就可能使用不正确的数据来执行。

在一般的编程语言中，函数的另一个行为是可以有返回值。类似地，不仅可以把变元传送给工作流，也可以从工作流中传出变元。前面创建了一个字典，以便把值传送给工作流，也可以用相同的方式在工作流结束时，从字典中返回输出变元。

变元包含方向的概念，方向可以是如下 3 个值中的一个：

- In: 变元传送到工作流中。
- Out: 从工作流返回变元。
- In/Out: 变元传送到工作流中，在工作流结束时从工作流中返回。

只有定义为 Out 或 In/Out 的变元才能在工作流结束时返回。为了读取从工作流返回的值，可以使用以下代码：

```
IDictionary<string,object> returnValues = WorkflowInvoker.Invoke(new Workflow1(),
    parms);
```

这里给 returnValues 变量赋予一个名称/值对字典，该字典包含工作流上定义的所有 Out 和 In/Out 变元。

下面的示例说明了如何将变元传给工作流以及如何从工作流传出变元。

## 试一试：返回变元

- (1) 在 Visual Studio 2010 中创建一个新 Workflow Console Application 项目。
- (2) 在工作流中创建一个字符串变元，把它定义为 In/Out 变元 Person。
- (3) 把一个 Sequence 活动拖放到工作流中。
- (4) 把一个 WriteLine 活动拖放到工作流中，把其文本表达式设置为：

```
String.Format ( "Person is called : {0}", Person )
```



这是 VB，并非 C#；这是一个表达式，所以不需要分号。

- (5) 把一个 Assign 活动拖放到工作流中，把其左边的文本框设置为 Person，右边的文本框设置为如下表达式：

```
String.Format("You entered the name : {0}", Person)
```

此时的工作流应如图 27-11 所示。

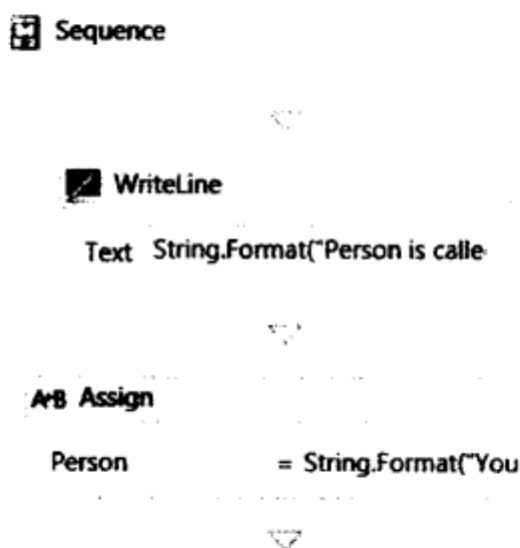


图 27-11

- (6) 修改主 program.cs 文件，使之把一个变元传送给工作流，并输出所有 Out 或 In/Out 变元的值，如下面的代码段所示：

```
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("Person", "Morgan");

foreach (KeyValuePair<string, object> kvp in
    WorkflowInvoker.Invoke(new Workflow1(), parms))
{
    Console.WriteLine("{0} = {1}", kvp.Key, kvp.Value);
}
```

执行时，工作流应输出 Person 变元的值，再从上述代码中输出一个修改过的值。这说明，一旦

workflows完成，就把 workflows中修改过的变元传送给调用者。

示例的说明

执行这个 workflows时，给它传送一个输入变元。在 workflows执行时，这个变元是可用的，在这个例子中， workflows结束时还会返回该变元，因为该变元定义为 In/Out。

27.4 定制活动

本章前面的示例仅使用了内置活动，而 workflows还允许编写定制活动，接着就可以像内置活动那样使用定制活动。

本章的前面提到，有两大类活动：单一活动和复合活动。本节将创建这两类活动。

活动由拥有它的工作流(或父活动)安排执行。接着要执行什么操作主要取决于活动的编写者。例如对于 WriteLine 活动，在该活动的代码中一般会调用 Console.WriteLine。

编写活动时，一般会重写 Execute 方法，以便提供定制代码。这个方法随活动的基类而变化。这些基类及其执行方法如表 27-1 所示。

表 27-1

基 类	执 行 方 法
AsyncCodeActivity	IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object) void EndExecute(AsyncCodeActivityContext, IAsyncResult)
CodeActivity	void Execute (CodeActivityContext)
NativeActivity	void Execute (NativeActivityContext)
AsyncCodeActivity<TResult>	IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object) TResult EndExecute(AsyncCodeActivityContext, IAsyncResult)
CodeActivity<TResult>	TResult Execute (CodeActivityContext)
NativeActivity<TResult>	void Execute (NativeActivityContext)

最简单的基类是 CodeActivity，CodeActivity 还有一个泛型版本，该版本接受一个类型变元——用作执行该活动时的返回值。工作流可以返回变元，同样，活动在执行后也可以返回一个值，这个数据可以在 workflows中绑定，这样，一个活动的输出就可以变成下一个活动的输入。

假定要在 workflows中使用当前时间，就可以创建一个返回 DateTime 值的活动，该活动执行时通过调用 DateTime.Now 获得这个时间信息。只要创建一个定制活动即可，而无需把一个字符串输出到控制台上。下面的示例说明了创建定制活动的过程。

试一试：编写定制活动

- (1) 在 Visual Studio 2010 中创建一个新的 Workflow Console Application 项目。
- (2) 在解决方案中添加第二个项目，但这个项目使用 Activity Library 项目模板。这会创建一个

默认活动(Activity1.xaml), 可以从项目中删除这个默认活动, 因为目前不使用它。

(3) 给类库添加一个新类 `Timestamp`, 需要的代码如下:

```
using System;
using System.Activities;
namespace CustomActivities
{
    public class Timestamp : CodeActivity<DateTime>
    {
        protected override DateTime Execute(CodeActivityContext context)
        {
            return DateTime.Now;
        }
    }
}
```

这段代码定义了定制活动, 并提供了对应 `Execute` 方法的实现代码(返回当前的日期/时间值)。

(4) 编译解决方案, 再把对定制活动项目的引用添加到 workflow 项目中。这样就可以在 workflow 项目中使用定制活动, 并把定制活动添加到工具箱中。

(5) 编辑主 workflow, 拖放一个 `Sequence` 活动。在 `Sequence` 活动中定义一个 `DateTime` 类型的变量, 把这个变量命名为 `currentDateTime`。

(6) 拖放一个 `Timestamp` 活动, 并显示其属性。需要把 `Result` 属性改为 `currentDateTime`, 将活动的结果值赋予这个变量。图 27-12 显示了属性值。

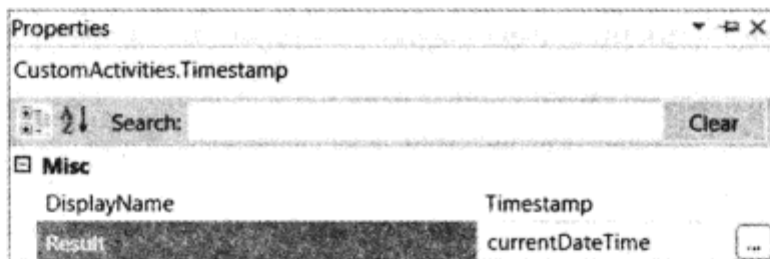


图 27-12

(7) 拖放一个 `WriteLine` 活动, 设置其表达式, 如下所示:

```
String.Format ( "The time read from the Timestamp activity is '{0}'",
    currentDateTime)
```

(8) 运行 workflow。结果应显示了当前的日期和时间。

### 示例的说明

执行 workflow 时, 会依次运行每个活动。如果活动派生自 `CodeActivity`, 在安排运行该活动时就会调用 `Execute` 方法。这里的活动有一个返回值, 该返回值在 `Execute` 方法中设置为当前的日期和时间。在这个示例中, `Timestamp` 活动的值存储在一个 workflow 变量中, 接着使用 `WriteLine` 活动输出到控制台上。

`Timestamp` 活动非常简单。一般情况下, 我们创建的活动要在其 `Execute` 方法中执行更多的操作。活动一般是自包含的单元, 类似于传统编程语言中的函数。函数一般有一个或多个变元, 通常这些变元作为参数传送给函数。但有时函数还通过当前的应用程序环境接收数据。



ASP.NET 中有一个很好的例子。静态属性 `HttpContext.Current` 允许访问各种属性，例如当前应用程序的状态、`HttpRequest` 和 `HttpResponse`。这个对象由 ASP.NET 处理管道定义，接着就可用于在该管道内部调用的任何对象。

在工作流中，使用“扩展”概念也可以实现类似的功能。

### 27.4.1 工作流扩展

扩展只是一个要从活动的 `Execute` 方法中访问的对象。一般需要为扩展定义一个接口，活动则针对这个接口进行编码。这样就可以替换该扩展的实现代码，而无需重新编写整个活动的代码。

例如，考虑一个发送电子邮件的活动。可以在活动内部硬编码电子邮件提供程序，但该活动只能使用这个提供程序。在这个例子中，最好定义一个活动使用的接口，再提供该接口的几个实现方式——其中一个使用 Outlook 发送邮件，另一个使用 Microsoft Exchange，等等。简言之，可以扩展电子邮件提供程序的列表，而无需修改活动。

为了使用电子邮件活动，还需要能定义传送给该活动的变元。我们肯定不希望硬编码电子邮件的接收人、主题或正文。与 `WriteLine` 活动一样，我们希望能定义可以在活动中设置的属性。为此，需要使用派生自 `Argument` 的类，例如 `InArgument` 和 `OutArgument`。

使用这些变元类替代活动中的属性。使用变元类型的原因与工作流在执行时存储其状态的方式相关，参见本章后面的内容。下面创建接口和定制活动。

#### 试一试：定义 `ISendEmail` 接口和活动

- (1) 在 Visual Studio 2010 中创建一个新 Class Library 项目，命名为 `SharedInterfaces`。
- (2) 在库中添加一个接口 `ISendEmail`，如下所示：

```
/// <summary>
/// Interface used by the SendEmail activity to send an email
/// </summary>
public interface ISendEmail
{
    /// <summary>
    /// Sends an email
    /// </summary>
    /// <param name="sender">The person sending the email</param>
    /// <param name="recipient">The recipient of the email</param>
    /// <param name="subject">The subject</param>
    /// <param name="body">The body</param>
    void SendEmail(string sender, string recipient, string subject, string body);
}
```

- (3) 现在给解决方案添加第二个项目。这次选择 Activity Library，命名为 `CustomActivities`。删除自动创建的 `Activity1.xaml` 文件。

- (4) 给项目添加一个新类 `SendEmail`，其定义如下所示。需要基于活动库引用 `SharedInterfaces` 项目，以便包含 `ISendEmail` 接口的定义：

```
public class SendEmail : NativeActivity
{
    public InArgument<string> Sender { get; set; }
    public InArgument<string> Recipient { get; set; }
```



```

public InArgument<string> Subject { get; set; }
public InArgument<string> Body { get; set; }
protected override void Execute(NativeActivityContext context)
{
    context.GetExtension<ISendEmail>().SendEmail
        (Sender.Get(context), Recipient.Get(context),
        Subject.Get(context), Body.Get(context));
}
}

```

该活动定义了 4 个输入变元，并在 Execute 方法中使用它们。

### 示例的说明

执行这个活动时，只是查找 ISendEmail 扩展，并调用其 SendEmail 方法。发送电子邮件还需要更多的功能，这是下面两个示例的主题。

定义变元时，应使用泛型类 InArgument<>、OutArgument<>或 InOutArgument<>。在 Execute 方法中，使用有点古怪的语法 Argument.Get(context)，从当前执行环境中检索这些变元的值。这是由数据在工作流中的存储方式决定的。

在使用常规.NET 属性的传统类中，该类的数据存储在对象实例中。这说明，数据对于外部的调用者而言是不透明的；而在工作流中包含活动，意味着要在磁盘上存储工作流实例，只有完全序列化每个活动，才能把工作流存储在磁盘上。

这是 Workflow 3.x 的工作方式，它导致一些工作流需要使用磁盘上的大量空间。Workflow 4 使用了新的模型，只需要存储发生了变化的数据，因为传送给活动的环境对象可以跟踪这些变元的实际值。在 SendEmail 活动中，使用 Get()方法从环境中读取数据，所以在这个活动执行时，工作流的状态会保存下来。例如，如果调用 Set()方法，改变变元的值，工作流执行逻辑就会设置一个标记，表示有数据改变了，以便仅把这些变化的数据存储到磁盘上。这极大地提高了 Workflow 4 的性能，磁盘的使用量也显著减少了。

除了维护所有变元的值之外，环境对象还包含一个扩展集合。在上面的代码中，可以调用 GetExtension<>泛型方法，从环境对象中检索 ISendEmail 接口。这里传送了我们请求的接口类型，这个方法中的查找逻辑给代码返回扩展实例，以便在该扩展上调用 SendEmail 方法。

如下面的示例所示，下一步是把扩展添加到工作流中，为此，需要利用工作程序集中的另一个类 WorkflowApplication。

### 试一试：使用 WorkflowApplication 类

(1) 给解决方案添加第三个项目。这应是一个 Workflow Console Application 项目。之后，把它设置为启动项目。

(2) 添加对 SharedInterfaces 和 CustomActivities 程序集的引用。

(3) 添加 ISendEmail 接口的实现代码，如下所示(这些代码不发送电子邮件，但至少把数据输出到控制台上)：

```

public class ConsoleSendEmail : ISendEmail
{
    public void SendEmail(string sender, string recipient, string subject,

```

```

        string body)
    {
        Console.WriteLine("Email to:      {0}", recipient);
        Console.WriteLine("      from:      {0}", sender);
        Console.WriteLine("      subject: {0}", subject);
        Console.WriteLine("      body:      {0}", body);
    }
}

```

(4) 在工作流中添加一个 SendEmail 活动，设置所有的属性。按 F4 键会显示所选活动的属性表。

(5) 修改 Program.cs 文件，使用 WorkflowApplication 类执行工作流。这个类允许添加扩展，而 WorkflowInvoker 则不然。

```

class Program
{
    static void Main(string[] args)
    {
        WorkflowApplication app = new WorkflowApplication(new Workflow1());
        app.Extensions.Add(new ConsoleSendEmail());
        ManualResetEvent finished = new ManualResetEvent(false);
        app.Completed = (completedArgs) => { finished.Set(); };
        app.Run();
        finished.WaitOne();
    }
}

```

这里创建了 WorkflowApplication 实例，接着给它添加 ConsoleSendEmail 扩展。然后创建 ManualResetEvent，给它关联 Completed 事件。工作流结束时会引发该事件。之后调用 Run 方法以执行工作流，在事件完成后，工作流就结束了。如果生成并运行程序，则控制台上显示的输出应匹配在 SendEmail 活动上设置的属性值。

### 示例的说明

执行工作流应用程序时，添加到应用程序上的扩展会存储在 WorkflowApplication 对象的集合中。该对象安排每个活动的执行，在执行一个活动时，会创建一个环境对象，并将其传送给 Execute 方法。

环境对象的类型随活动使用的基类而变化。由于 SendEmail 活动派生自 NativeActivity，所以可以调用 GetExtension 方法，以检索添加到工作流应用程序中的任何扩展。

如果活动派生自 CodeActivity，传送给其 Execute 方法的环境对象就不能访问任何扩展，实际上，根据设计，CodeActivity 不能访问任何环境信息，

在工作流应用程序上调用 Run 时，会使用线程池中的线程来执行工作流，因此在执行工作流时，允许代码在后台继续执行。上面的代码使用 ManualResetEvent，并在处理程序中给 Completed 事件设置了 ManualResetEvent，使主应用程序与工作流的完成同步。

测试了前面的代码后，就可以确认 SendEmail 活动在工作，所以现在可以使用 System.Net.Mail 名称空间中的类创建 ISendEmail 的实际实现代码。理解这个名称空间的一个很好的资源是 [www.systemnetmail.com](http://www.systemnetmail.com)。

另一种方法是用 Outlook 发送电子邮件，下一个例子就完成这个功能。

试一试：用 Outlook 发送电子邮件

(1) 给解决方案添加第三个项目。这应是一个 Workflow Console Application 项目。之后，把它设置为启动项目。

(2) 添加对 Outlook 对象模型的引用。为此，打开 Add Reference 对话框，选择 COM 选项卡。再向下滚动，找到 Microsoft Outlook Object Library。用于演示这个示例的计算机安装了 Microsoft Office 2007，其内部的版本号是 12.0，所以它会显示在对话框中，如图 27-13 所示。

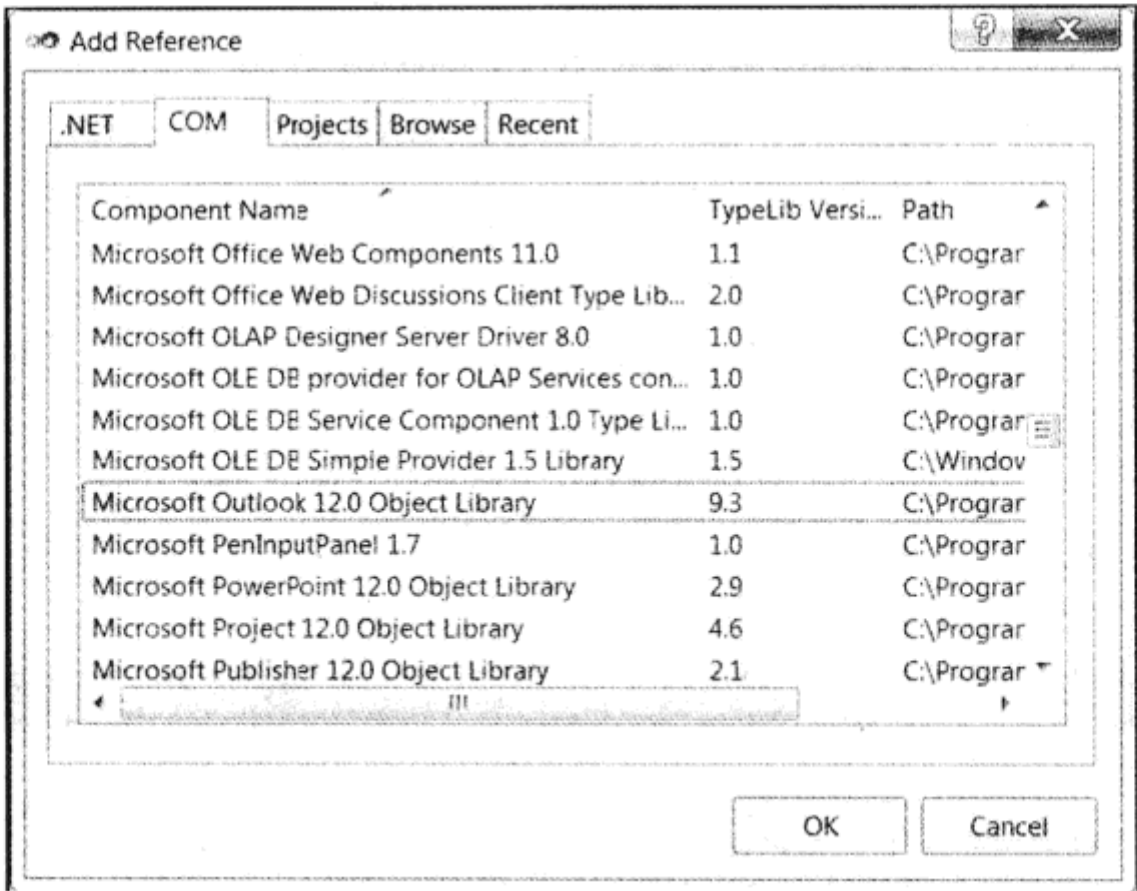


图 27-13

(3) 创建一个新类 OutlookSendEmail，键入如下代码：

```
public class OutlookSendEmail : ISendEmail
{
    public void SendEmail(string sender, string recipient, string subject,
        string body)
    {
        Application app = new Application();
        var mapi = app.GetNamespace("MAPI");
        mapi.Logon(ShowDialog: false, NewSession: false);
        var outbox = mapi.GetDefaultFolder(OlDefaultFolders.olFolderOutbox);

        MailItem email = app.CreateItem(OlItemType.olMailItem);
        email.To = recipient;
        email.Subject = subject;
        email.Body = body;
        email.Send();
    }
}
```

注意在这个示例中，没有指定发件人，因为电子邮件使用当前登录的用户的配置信息来发送。

(4) 修改 Program.cs 文件, 把这个类用作电子邮件扩展:

```
app.Extensions.Add(new OutlookSendEmail());
```

应删除 ConsoleSendEmail 扩展, 或者将其注释掉。

(5) 运行程序(确保 Outlook 也在运行)。如果查看 Sent Items 文件夹, 应看到自动生成的电子邮件。这说明我们刚才发送了第一封自动生成的电子邮件。

#### 示例的说明

OutlookSendEmail 类使用 Outlook 对象模型, 通过 MailItem 类新建一封电子邮件。为了用 Outlook 发送电子邮件, 需要构建 Application 对象的一个实例, 再获取 MAPI 名称空间的引用。

如果尚未运行 Outlook, 就需要在对 Login 的调用中指定用户名和密码。如果已在运行 Outlook, 就可以忽略这些参数, 此时 Outlook 将使用当前登录的用户的配置信息。

创建了 MailItem 后, 就可以设置 To 属性, 以指定收件人。要发送给多个收件人, 可以在电子邮件地址之间添加分号。然后需要指定电子邮件的 Body 和 Subject, 最后调用 Send()。如果希望发送格式化的电子邮件, 也可以使用 HTMLBody 属性。

如果在发件箱中看不到电子邮件, 或者从代码中接收到异常, 就需要跟踪这个异常。为此, 可以在 Program.cs 文件中添加一些额外的代码, 如下面的示例所示。

#### 试一试: 处理工作流的错误

在这个例子中, 将学习如何捕获和处理工作流中的错误。

(1) 使用上一个例子的项目, 修改 Program.cs 文件, 如下所示:

```
static void Main(string[] args)
{
    WorkflowApplication app = new WorkflowApplication(new Workflow1());
    app.OnUnhandledException = (e) =>
    {
        return UnhandledExceptionAction.Abort;
    };
    app.Extensions.Add(new OutlookSendEmail());
    ManualResetEvent finished = new ManualResetEvent(false);
    app.Completed = (completedArgs) => { finished.Set(); };
    app.Aborted = (abortedEventArgs) =>
    {
        Console.WriteLine("Workflow Aborted.\r\n{0}", abortedEventArgs.Reason);
        finished.Set();
    };
    app.Run();
    finished.WaitOne();
}
```

已经添加了突出显示的条目。

(2) 运行应用程序。如果出现了异常, 就显示一个消息 Workflow Aborted, 然后把异常输出到控制台上。

### 示例的说明

工作流中出现未处理的异常时，首先调用的是 `OnUnhandledException` 委托。这里可以选择执行的动作有 `Abort`、`Cancel` 或 `Terminate`。这个委托传送给异常的一个实例，以便根据所抛出异常的类型确定执行什么操作。

如果选择 `Abort`(中止)工作流，就接着调用 `Aborted` 委托。默认是 `Terminate`(终止)工作流。

## 27.4.2 活动的有效性验证

如果没有定义变元，许多活动都不能执行，目前我们无法把给定的变元标记为必选变元。使用一些标准活动时，可能会在工作流设计器上显示一个错误消息，因为这些活动包含一些必选变元。

为了把属性标记为必选属性，可以在定义变元时使用 `[RequiredArgument]` 特性。在给变元添加这个特性时，在活动的右边会显示一个红色的感叹号图标，如图 27-14 所示。

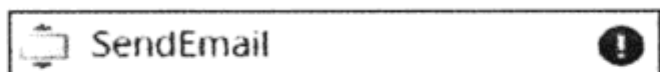


图 27-14

这表示一个或多个属性存在错误，如果把光标停放在该图标上，会显示一个描述该错误的工具提示。下面的示例将更新 `SendMail` 活动，把除 `Sender` 之外的所有变元标记为必选变元。

### 试一试：把变元标记为必选变元

(1) 打开 `SendEmail.cs` 文件，进行如下修改：

```
public class SendEmail : NativeActivity
{
    public InArgument<string> Sender { get; set; }
    [RequiredArgument]
    public InArgument<string> Recipient { get; set; }
    [RequiredArgument]
    public InArgument<string> Subject { get; set; }
    [RequiredArgument]
    public InArgument<string> Body { get; set; }
    protected override void Execute(NativeActivityContext context)
    {
        context.GetExtension<ISendEmail>().SendEmail(Sender.Get(context),
            Recipient.
            Get(context), Subject.Get(context), Body.Get(context));
    }
}
```

(2) 编译应用程序。

(3) 打开 `Workflow1.xaml` 文件，显示 `SendEmail` 活动的属性。修改一个加了 `[RequiredArgument]` 特性的属性，使光标退出其文本框。此时应看到错误图标，把鼠标停放在该图标上，查看错误的描述。

(4) 在必选变元中输入一个值，再次使光标退出其文本框，错误消息应消失了。

### 示例的说明

工作流可以检查活动，查找 `RequiredArgument` 特性。如果找到了带这个特性的属性，但该属性

没有定义其值，设计器类就会给活动加上错误图标。

定制活动就快完成了。最后一个任务是创建一个定制设计器，用于提供活动在设计期间可视化表示。

### 27.4.3 活动设计器

把活动拖放到设计界面上时，设计器会提供其可视化表示。一般这是一个 Windows 窗体类，而在 Visual Studio 2010 中，可以使用 XAML 定义活动的设计器。

XAML 详见第 22 章，这里不再重复。本节仅讨论对定制活动比较重要的部分。

一般在独立的程序集中创建活动的设计器类，它仅需要设计期间使用，在执行活动时不需要它。Visual Studio 2010 包含 Activity Designer Library 项目类型，提供了足以开始创建设计器的功能，下面的例子就使用这个项目类型。

除了提供活动的可视化表示之外，设计器还可以提供数据输入字段。没有设计器，活动的所有属性都必须在属性表中设置；而对于定制设计器，可以选择在设计界面上包含一些属性。这可以为活动的用户提供非常棒的设计期间的体验。

下面的示例将再次更新 SendEmail 活动，添加一个定制设计器。

#### 试一试：添加活动设计器

(1) 打开前面的解决方案，再添加一个新的 Activity Designer Library 项目，命名为 CustomActivities.Design。

(2) 这会创建一个空白的设计器 ActivityDesigner1。可以重命名这个设计器，或者添加一个新的设计器 SendEmailDesigner，无论采用哪种方式，都应有一个名称与活动名相似的设计器。

(3) 默认创建的 XAML 提供了一个空白的设计界面，在其中需要添加一些文本字段和标签。在设计器的 XAML 文件中添加如下 XAML——这定义了一组列和行，在这些列和行中将放置设计元素：

```
<sap:ActivityDesigner x:Class="CustomActivities.Design.SendEmailDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sap="clr-namespace:System.Activities.Presentation;
    assembly=System.Activities.Presentation"
    xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
    assembly=System.Activities.Presentation">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
    </Grid>
</sap:ActivityDesigner>
```



(4) 在屏幕上添加接收用户输入的元素:

```
</Grid.RowDefinitions>
<TextBlock Text="Recipient"/>
<TextBox Text="{Binding ModelItem.Recipient}" Grid.Column="1"/>
<TextBlock Text="Subject" Grid.Row="1"/>
<TextBox Text="{Binding ModelItem.Subject}" Grid.Row="1" Grid.Column="1"/>
<TextBlock Text="Body" Grid.Row="2"/>
<TextBox Text="{Binding ModelItem.Body}" Grid.Row="2" Grid.Column="1"/>
</Grid>
```

这些元素定义了一系列标签和文本框, 它们数据绑定到底层的活动上——前缀 `ModelItem` 是实际活动的同义词。

(5) 需要把设计器与活动关联起来。最简单的方法是使用 `Designer` 特性。在 `SendEmail` 活动的顶部添加如下代码:

```
using System.ComponentModel;
namespace CustomActivities
{
    [Designer("CustomActivities.Design.SendEmailDesigner,
    CustomActivities.Design")]
    public class SendEmail : NativeActivity
    {
```

`Designer` 特性由 Visual Studio 读取, 用于确定把哪个设计器关联到活动上, 以及哪个程序集包含设计器。上面使用的字符串是设计器的 `TypeName`, 它一般输入为字符串, 以免在设计器程序集和活动程序集之间进行程序集引用。

(6) 在主工作程序集中添加对 `PresentationCore` 程序集的引用。如果接着编译解决方案, 打开包含 `SendEmail` 活动的工作流, 将看到如图 27-15 所示的结果。



图 27-15

(7) 这个设计器功能完备, 但不是很吸引人, 为此可以在字段周围添加一些空格。修改 XAML, 如下所示, 就会得到比较好的设计结果。当然也可以添加色彩和图形, 使设计界面更加鲜活。

```
<Grid>
  <Grid.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="Margin" Value="0,2,4,2"/>
      <Setter Property="VerticalAlignment" Value="Center"/>
    </Style>
    <Style TargetType="TextBox">
      <Setter Property="Margin" Value="0,2,0,2"/>
    </Style>
  </Grid.Resources>
  <Grid.ColumnDefinitions>
```

这些资源定义了与文本块和文本框关联的样式。这里的这些样式仅应用了统一的页边距和对齐方式, 使活动在屏幕上看起来更加美观。



示例的说明

Visual Studio 使用 Designer 特性查找与活动关联的类。如果找到了，就在屏幕上显示活动时使用该类。

在 XAML 中常常使用数据绑定把可视化类和后台类链接起来——在这个例子中，可视化类是设计器，后台类是活动。

27.5 小结

本章学习了 Windows Workflow Foundation 4，主要内容如下：

- 工作流的含义和执行方式
- 如何使用一些内置活动
- 如何创建自己的活动

27.6 练习

- (1) 如何创建复合活动？
- (2) 可以通过 WCF 展示工作流吗？如果能，如何展示？
- (3) 如何确保工作流从停止的地方重新启动？

附录 A 给出了练习答案。

27.7 本章要点

主 题	重 要 概 念
工作流基础	工作流由活动组成，活动类似于传统编程语言中的语句。可以编写自己的活动。工作流一般包含一些内置的活动和一些定制的活动
If 活动	可以在工作流中使用 If 活动计算一个表达式，选择两条路径中的一条。表达式可以很简单，也可以很复杂，可以根据需要引用变量和变元
While 活动	这个活动允许在工作流中定义一个循环。循环的条件是一个表达式，该活动包含一个子活动，该子活动一般是一个序列，这样就可以在循环的每次迭代中添加其他多个活动
Sequence 活动	Sequence 活动允许自上而下地执行许多子活动
变元和变量	可以在工作流中传入传出变元，在工作流中还可以定义全局或本地变量。变元通过数据类型来定义，例如 String 或 Int32，还需要定义变元的方向。变量遵守的规则与传统的编程语言相同
工作流扩展	扩展可以用于改变工作流在运行期间的行为，而无需改变工作流。扩展一般编写为接口和该接口的实现代码
活动的有效性验证	可以把活动的一些属性定义为必选属性，这样终端用户就可以看出哪些属性必须定义其值，而没有定义其值的必选属性会在用户界面上显示错误图标
活动设计器	设计器可以用于美化活动的用户界面，使终端用户更容易使用活动。设计器是 XAML，可以创建任何标记，来显示定制活动的用户界面

# 附录 A

## 习题答案

第 1、2 章没有习题。

### 第 3 章

#### 习题 1

`super.smashing.great`

#### 习题 2

b), 因为它以数字开头; e), 因为它包含一个句点。

#### 习题 3

不, 理论上没有限制包含在 `string` 变量中的字符串的长度。

#### 习题 4

这里, `*`和`/`运算符的优先级最高, 其次是`+`, `<<`, 最后是`+=`。本习题中的优先级可以用括号来演示, 如下所示:

```
resultVar += (var1 * var2) + (var3 % (var4 / var5));
```

或者:

```
resultVar += (var1 * var2) + ((var3 % var4) / var5));
```

二者的结果相同。

#### 习题 5

```
static void Main(string[] args)
{
    int firstNumber, secondNumber, thirdNumber, fourthNumber;
```

```

    Console.WriteLine("Give me a number:");
    firstNumber = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Give me another number:");
    secondNumber = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Give me another number:");
    thirdNumber = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Give me another number:");
    fourthNumber = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("The product of {0}, {1}, {2}, and {3} is {4}.",
        firstNumber, secondNumber, thirdNumber, fourthNumber,
        firstNumber * secondNumber * thirdNumber * fourthNumber);
}

```

注意这里使用了 Convert.ToInt32(), 本章没有介绍它们。

## 第 4 章

### 习题 1

```
(var1 > 10) ^ (var2 > 10)
```

### 习题 2

```

static void Main(string[] args)
{
    bool numbersOK = false;
    double var1, var2;
    var1 = 0;
    var2 = 0;
    while (!numbersOK)
    {
        Console.WriteLine("Give me a number:");
        var1 = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Give me another number:");
        var2 = Convert.ToDouble(Console.ReadLine());
        if ((var1 > 10) ^ (var2 > 10))
        {
            numbersOK = true;
        }
        else
        {
            if ((var1 <= 10) && (var2 <= 10))
            {
                numbersOK = true;
            }
            else
            {
                Console.WriteLine("Only one number may be greater than 10.");
            }
        }
    }
    Console.WriteLine("You entered {0} and {1}.", var1, var2);
}

```

注意使用另一个逻辑时, 执行效果更佳, 如下所示:

```
static void Main(string[] args)
{
    bool numbersOK = false;
    double var1, var2;
    var1 = 0;
    var2 = 0;
    while (!numbersOK)
    {
        Console.WriteLine("Give me a number:");
        var1 = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Give me another number:");
        var2 = Convert.ToDouble(Console.ReadLine());
        if ((var1 > 10) && (var2 > 10))
        {
            Console.WriteLine("Only one number may be greater than 10.");
        }
        else
        {
            numbersOK = true;
        }
    }
    Console.WriteLine("You entered {0} and {1}.", var1, var2);
}
```

### 习题 3

代码如下:

```
int i;
for (i = 1; i <= 10; i++)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine(i);
}
```

使用赋值运算符=而不是布尔运算符==, 是一个十分常见的错误。

### 习题 4

```
static void Main(string[] args)
{
    double realCoord, imagCoord;
    double realMax = 1.77;
    double realMin = -0.6;
    double imagMax = -1.2;
    double imagMin = 1.2;
    double realStep;
    double imagStep;
    double realTemp, imagTemp, realTemp2, arg;
    int iterations;
    while (true)
    {
```

```

    realStep = (realMax - realMin) / 79;
    imagStep = (imagMax - imagMin) / 48;
    for (imagCoord = imagMin; imagCoord <= imagMax;
        imagCoord += imagStep)
    {
        for (realCoord = realMin; realCoord <= realMax;
            realCoord += realStep)
        {
            iterations = 0;
            realTemp = realCoord;
            imagTemp = imagCoord;
            arg = (realCoord * realCoord) + (imagCoord * imagCoord);
            while ((arg < 4) && (iterations < 40))
            {
                realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
                    - realCoord;
                imagTemp = (2 * realTemp * imagTemp) - imagCoord;
                realTemp = realTemp2;
                arg = (realTemp * realTemp) + (imagTemp * imagTemp);
                iterations += 1;
            }
            switch (iterations % 4)
            {
                case 0:
                    Console.WriteLine(".");
                    break;
                case 1:
                    Console.WriteLine("o");
                    break;
                case 2:
                    Console.WriteLine("O");
                    break;
                case 3:
                    Console.WriteLine("@");
                    break;
            }
        }
        Console.WriteLine("\n");
    }
    Console.WriteLine("Current limits:");
    Console.WriteLine("realCoord: from {0} to {1}", realMin, realMax);
    Console.WriteLine("imagCoord: from {0} to {1}", imagMin, imagMax);

    Console.WriteLine("Enter new limits:");
    Console.WriteLine("realCoord: from:");
    realMin = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("realCoord: to:");
    realMax = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("imagCoord: from:");
    imagMin = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("imagCoord: to:");
    imagMax = Convert.ToDouble(Console.ReadLine());
}
}

```

## 第 5 章

### 习题 1

a)和 c)的转换不能隐式进行。

### 习题 2

```
enum color : short
{
    Red, Orange, Yellow, Green, Blue, Indigo, Violet, Black, White
}
```

可以, byte 类型可以包含 0~255 之间的数字。如果枚举项使用不同值, 基于 byte 的枚举可以包含 256 项, 如果给枚举项使用重复的值, 就可以包含更多的项。

### 习题 3

```
static void Main(string[] args)
{
    imagNum coord, temp;
    double realTemp2, arg;
    int iterations;
    for (coord.imag = 1.2; coord.imag >= -1.2; coord.imag -= 0.05)
    {
        for (coord.real = -0.6; coord.real <= 1.77; coord.real += 0.03)
        {
            iterations = 0;
            temp.real = coord.real;
            temp.imag = coord.imag;
            arg = (coord.real * coord.real) + (coord.imag * coord.imag);
            while ((arg < 4) && (iterations < 40))
            {
                realTemp2 = (temp.real * temp.real) - (temp.imag * temp.imag)
                    - coord.real;
                temp.imag = (2 * temp.real * temp.imag) - coord.imag;
                temp.real = realTemp2;
                arg = (temp.real * temp.real) + (temp.imag * temp.imag);
                iterations += 1;
            }
            switch (iterations % 4)
            {
                case 0:
                    Console.Write(".");
                    break;
                case 1:
                    Console.Write("o");
                    break;
                case 2:
                    Console.Write("O");
                    break;
                case 3:
                    Console.Write("@");
```

```

        break;
    }
}
Console.WriteLine("\n");
}
}

```

#### 习题 4

不, 原因如下:

- 遗漏了语句末尾的分号。
- 第二行尝试访问 blab 中不存在的第 6 个元素。
- 第二行尝试指定未包含在双引号中的字符串。

#### 习题 5

```

static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
    string reversedString = "";
    for (int index = myString.Length - 1; index >= 0; index--)
    {
        reversedString += myString[index];
    }
    Console.WriteLine("Reversed: {0}", reversedString);
}

```

#### 习题 6

```

static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
    myString = myString.Replace("no", "yes");
    Console.WriteLine("Replaced \"no\" with \"yes\": {0}", myString);
}

```

#### 习题 7

```

static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();
    myString = "\"" + myString.Replace(" ", "\" \") + "\"";
    Console.WriteLine("Added double quotes around words: {0}", myString);
}

```

或者使用 String.Split():

```

static void Main(string[] args)
{
    Console.WriteLine("Enter a string:");
    string myString = Console.ReadLine();

```



```

    string[] myWords = myString.Split(' ');
    Console.WriteLine("Adding double quotes around words:");
    foreach (string myWord in myWords)
    {
        Console.WriteLine("\"{0}\" ", myWord);
    }
}

```

## 第 6 章

### 习题 1

第一个函数的返回类型是 bool，但不返回一个 bool 值。

第二个函数有一个 params 变元，但这个变元不在变元列表的末尾。

### 习题 2

```

static void Main(string[] args)
{
    if (args.Length != 2)
    {
        Console.WriteLine("Two arguments required.");
        return;
    }
    string param1 = args[0];
    int param2 = Convert.ToInt32(args[1]);
    Console.WriteLine("String parameter: {0}", param1);
    Console.WriteLine("Integer parameter: {0}", param2);
}

```

注意这个答案包含的代码检查是否提供了两个变元，本题没有这个要求，但在本题中似乎应进行检查。

### 习题 3

```

class Program
{
    delegate string ReadLineDelegate();

    static void Main(string[] args)
    {
        ReadLineDelegate readLine = new ReadLineDelegate(Console.ReadLine);
        Console.WriteLine("Type a string:");
        string userInput = readLine();
        Console.WriteLine("You typed: {0}", userInput);
    }
}

```

### 习题 4

```

struct order
{
    public string itemName;
}

```

```

    public int unitCount;
    public double unitCost;

    public double TotalCost()
    {
        return unitCount * unitCost;
    }
}

```

## 习题 5

```

struct order
{
    public string itemName;
    public int unitCount;
    public double unitCost;

    public double TotalCost()
    {
        return unitCount * unitCost;
    }

    public string Info()
    {
        return "Order information: " + unitCount.ToString() + " " + itemName +
            " items at $" + unitCost.ToString() + " each, total cost $" +
            TotalCost().ToString();
    }
}

```

## 第 7 章

### 习题 1

这个语句仅对要用于所有版本的信息有效。而且，我们常常希望仅在使用调试版本时输出调试信息。此时，应首选 `Debug.WriteLine()` 版本。

使用 `Debug.WriteLine()` 版本还有一个优点，该版本不会编译到发布版本中，从而使最终代码文件变得更小。

### 习题 2

```

static void Main(string[] args)
{
    for (int i = 1; i < 10000; i++)
    {
        Console.WriteLine("Loop cycle {0}", i);
        if (i == 5000)
        {
            Console.WriteLine(args[999]);
        }
    }
}

```



在 VS 中，可以把断点放在下面的代码行上：

```
Console.WriteLine("Loop cycle {0}", i);
```

应修改断点的属性，把执行次数的条件设置为“执行次数等于 5000 时中断”。

在 VCE 中，可以把断点放在出错的代码行上，因为在 VCE 中无法采用上述方式修改断点的属性。

### 习题 3

错误，finally 块总是执行，它可能在处理 catch 块之后执行。

### 习题 4

```
static void Main(string[] args)
{
    Orientation myDirection;
    for (byte myByte = 2; myByte < 10; myByte++)
    {
        try
        {
            myDirection = checked((Orientation)myByte);
            if ((myDirection < Orientation.North) ||
                (myDirection > Orientation.West))
            {
                throw new ArgumentOutOfRangeException("myByte", myByte,
                    "Value must be between 1 and 4");
            }
        }
        catch (ArgumentOutOfRangeException e)
        {
            // If this section is reached then myByte < 1 or myByte > 4.
            Console.WriteLine(e.Message);
            Console.WriteLine("Assigning default value, Orientation.North.");
            myDirection = Orientation.North;
        }

        Console.WriteLine("myDirection = {0}", myDirection);
    }
}
```

注意这是一个小问题。因为枚举基于 byte 类型，所以可以给它赋予任何 byte 值，即使该值不是枚举中指定的名称，也是如此。在上面的代码中，如有必要，我们会生成自己的异常。

## 第 8 章

### 习题 1

Public、private 和 protected 是实际的可访问级别。

习题 2

错误，永远都不应手工调用对象的析构函数，.NET 运行库环境会在垃圾回收过程中自动完成该任务。

习题 3

不，可以在没有任何类实例的情况下调用静态方法。

习题 4

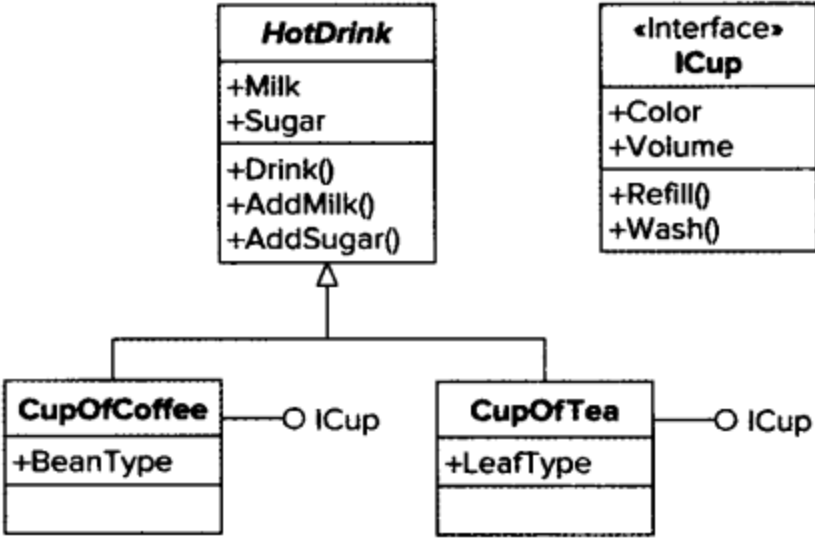


图 A-1

习题 5

```
static void ManipulateDrink(HotDrink drink)
{
    drink.AddMilk();
    drink.Drink();
    ICup cupInterface = (ICup)drink;
    cupInterface.Wash();
}
```

注意显式转换为 ICup 的代码行。这是必须的，因为 HotDrink 不支持 ICup 接口，但我们知道传送给这个函数的两个 cup 对象支持 ICup 接口。这很危险，因为也可以给这个函数传送其他类，但这类也可能派生于 HotDrink，而 HotDrink 却不支持 ICup 接口。为了更正这个问题，应检查该接口是否得到支持：

```
static void ManipulateDrink(HotDrink drink)
{
    drink.AddMilk();
    drink.Drink();
    if (drink is ICup)
    {
        ICup cupInterface = drink as ICup;
        cupInterface.Wash();
    }
}
```

这里使用的 `is` 和 `as` 操作符在第 11 章介绍。

## 第 9 章

### 习题 1

`myDerivedClass` 派生于 `MyClass`, 但是 `MyClass` 密封的, 不能从 `MyClass` 中派生其他类。

### 习题 2

把它定义为静态类, 或者将其所有构造函数定义为私有。

### 习题 3

不能创建的类可以通过它们拥有的静态成员来使用。实际上, 甚至可以通过这些成员获取这些类的实例, 如下所示:

```
class CreateMe
{
    private CreateMe()
    {
    }

    static public CreateMe GetCreateMe()
    {
        return new CreateMe();
    }
}
```

这里, 公共构造函数可以访问私有构造函数, 因为它在同一个类的定义中。

### 习题 4

为简单起见, 下面的类定义显示为一个代码文件的一部分, 而没有给每个类定义列出单独的代码文件:

```
namespace Vehicles
{
    public abstract class Vehicle
    {
    }
    public abstract class Car : Vehicle
    {
    }
    public abstract class Train : Vehicle
    {
    }
    public interface IPassengerCarrier
    {
    }
    public interface IHeavyLoadCarrier
    {
    }
}
```

```

    }
    public class SUV : Car, IPassengerCarrier
    {
    }
    public class Pickup : Car, IPassengerCarrier, IHeavyLoadCarrier
    {
    }
    public class Compact : Car, IPassengerCarrier
    {
    }
    public class PassengerTrain : Train, IPassengerCarrier
    {
    }
    public class FreightTrain : Train, IHeavyLoadCarrier
    {
    }
    public class T424DoubleBogey : Train, IHeavyLoadCarrier
    {
    }
}

```

## 习题 5

```

using System;
using Vehicles;

namespace Traffic
{
    class Program
    {
        static void Main(string[] args)
        {
            AddPassenger(new Compact());
            AddPassenger(new SUV());
            AddPassenger(new Pickup());
            AddPassenger(new PassengerTrain());
        }

        static void AddPassenger(IPassengerCarrier Vehicle)
        {
            Console.WriteLine(Vehicle.ToString());
        }
    }
}

```

## 第 10 章

### 习题 1

```

class MyClass
{
    protected string myString;
}

```



```

    public string ContainedString
    {
        set
        {
            myString = value;
        }
    }

    public virtual string GetString()
    {
        return myString;
    }
}

```

## 习题 2

```

class MyDerivedClass : MyClass
{
    public override string GetString()
    {
        return base.GetString() + " (output from derived class)";
    }
}

```

## 习题 3

如果方法有返回类型，就可以将其用作表达式的一部分：

```
x = Manipulate(y, z);
```

如果没有给部分方法提供实现代码，编译器就会在使用该部分方法的所有地方删除该方法。在上面的代码中，这会使 `x` 的结果变得模糊，因为 `Manipulate()` 方法没有替代方法。如果没有这个方法，可能只需忽略整行代码，但编译器无法确定我们是否的确希望忽略它。

没有返回类型的方法不能作为表达式的一部分来调用，所以编译器可以安全地删除对部分方法调用的所有引用。

同样，也禁止使用 `out` 参数，因为在方法调用之前，用作 `out` 参数的变量必须是未定义的，而应在方法调用之后定义。删除方法定义会违反这个规则。

## 习题 4

```

class MyCopyableClass
{
    protected int myInt;

    public int ContainedInt
    {
        get
        {
            return myInt;
        }
        set
        {
            myInt = value;
        }
    }
}

```



```

    }
}

public MyCopyableClass GetCopy()
{
    return (MyCopyableClass)MemberwiseClone();
}
}

```

客户端代码:

```

class Program
{
    static void Main(string[] args)
    {
        MyCopyableClass obj1 = new MyCopyableClass();
        obj1.ContainedInt = 5;
        MyCopyableClass obj2 = obj1.GetCopy();
        obj1.ContainedInt = 9;
        Console.WriteLine(obj2.ContainedInt);
    }
}

```

这些代码显示 5, 说明所复制对象有自己专用的 myInt 字段。

## 习题 5

```

using System;
using Ch10CardLib;

namespace Exercise_Answers
{
    class Class1
    {
        static void Main(string[] args)
        {
            while(true)
            {
                Deck playDeck = new Deck();
                playDeck.Shuffle();
                bool isFlush = false;
                int flushHandIndex = 0;
                for (int hand = 0; hand < 10; hand++)
                {
                    isFlush = true;
                    Suit flushSuit = playDeck.GetCard(hand * 5).suit;
                    for (int card = 1; card < 5; card++)
                    {
                        if (playDeck.GetCard(hand * 5 + card).suit != flushSuit)
                        {
                            isFlush = false;
                        }
                    }
                    if (isFlush)
                    {

```

```

        flushHandIndex = hand * 5;
        break;
    }
}
if (isFlush)
{
    Console.WriteLine("Flush!");
    for (int card = 0; card < 5; card++)
    {
        Console.WriteLine(playDeck.GetCard(flushHandIndex + card));
    }
}
else
{
    Console.WriteLine("No flush.");
}
Console.ReadLine();
}
}
}
}

```

这些代码会循环下去，因为同花色是不常见的。可能需要按几次回车键，才能在洗好的扑克牌中找到一个同花色。为了验证一切像期望的那样执行，可以尝试将洗牌的代码行注释掉。

## 第 11 章

### 习题 1

```

using System;
using System.Collections;

namespace Exercise_Answers
{
    public class People : DictionaryBase
    {
        public void Add(Person newPerson)
        {
            Dictionary.Add(newPerson.Name, newPerson);
        }

        public void Remove(string name)
        {
            Dictionary.Remove(name);
        }

        public Person this[string name]
        {
            get
            {
                return (Person)Dictionary[name];
            }
        }
    }
}

```

```
        set
        {
            Dictionary[name] = value;
        }
    }
}
```

## 习题 2

```
public class Person
{
    private string name;
    private int age;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            age = value;
        }
    }

    public static bool operator >(Person p1, Person p2)
    {
        return p1.Age > p2.Age;
    }

    public static bool operator <(Person p1, Person p2)
    {
        return p1.Age < p2.Age;
    }

    public static bool operator >=(Person p1, Person p2)
    {
        return !(p1 < p2);
    }

    public static bool operator <=(Person p1, Person p2)
```

```

    {
        return !(p1 > p2);
    }
}

```

### 习题 3

```

public Person[] GetOldest()
{
    Person oldestPerson = null;
    People oldestPeople = new People();
    Person currentPerson;
    foreach (DictionaryEntry p in Dictionary)
    {
        currentPerson = p.Value as Person;
        if (oldestPerson == null)
        {
            oldestPerson = currentPerson;
            oldestPeople.Add(oldestPerson);
        }
        else
        {
            if (currentPerson > oldestPerson)
            {
                oldestPeople.Clear();
                oldestPeople.Add(currentPerson);
                oldestPerson = currentPerson;
            }
            else
            {
                if (currentPerson >= oldestPerson)
                {
                    oldestPeople.Add(currentPerson);
                }
            }
        }
    }
    Person[] oldestPeopleArray = new Person[oldestPeople.Count];
    int copyIndex = 0;
    foreach (DictionaryEntry p in oldestPeople)
    {
        oldestPeopleArray[copyIndex] = p.Value as Person;
        copyIndex++;
    }
    return oldestPeopleArray;
}

```

这个函数比较复杂，因为没有为 `Person` 定义 `=` 运算符，但仍可以构建逻辑。另外，返回 `People` 实例更加简单，因为在处理过程中比较容易操作这个类。作为一个折中方法，在整个函数中都使用了 `People` 实例，再在最后转换为一个 `Person` 实例数组。

### 习题 4

```

public class People : DictionaryBase, ICloneable

```

```

{
    public object Clone()
    {
        People clonedPeople = new People();
        Person currentPerson, newPerson;
        foreach (DictionaryEntry p in Dictionary)
        {
            currentPerson = p.Value as Person;
            newPerson = new Person();
            newPerson.Name = currentPerson.Name;
            newPerson.Age = currentPerson.Age;
            clonedPeople.Add(newPerson);
        }
        return clonedPeople;
    }

    ...
}

```

在 Person 类上实现 ICloneable 接口，可以简化这段代码。

## 习题 5

```

public IEnumerable Ages
{
    get
    {
        foreach (object person in Dictionary.Values)
            yield return (person as Person).Age;
    }
}

```

## 第 12 章

### 习题 1

a、b 和 e 是  
c 和 d 否，但它们可以使用由包含它们的类提供的泛型类型参数。  
f 否。

### 习题 2

```

public static double? operator *(Vector op1, Vector op2)
{
    try
    {
        double angleDiff = (double)(op2.ThetaRadians.Value -
            op1.ThetaRadians.Value);
        return op1.R.Value * op2.R.Value * Math.Cos(angleDiff);
    }
    catch
    {
    }
}

```

```

        return null;
    }
}

```

### 习题 3

不在 T 上强制 new() 约束, 就不能实例化 T。在 T 上强制 new() 约束可以确保有一个公共的默认构造函数是可用的。

```

public class Instantiator<T>
    where T : new()
{
    public T instance;

    public Instantiator()
    {
        instance = new T();
    }
}

```

### 习题 4

同一个泛型类型参数 T 既用于泛型类, 又用于泛型方法。需要重命名其中的一个或两个。例如:

```

public class StringGetter<U>
{
    public string GetString<T>(T item)
    {
        return item.ToString();
    }
}

```

### 习题 5

一种方式如下:

```

public class ShortCollection<T> : IList<T>
{
    protected Collection<T> innerCollection;
    protected int maxSize = 10;

    public ShortCollection() : this(10)
    {
    }

    public ShortCollection(int size)
    {
        maxSize = size;
        innerCollection = new Collection<T>();
    }

    public ShortCollection(List<T> list) : this(10, list)
    {
    }
}

```

```
public ShortCollection(int size, List<T> list)
{
    maxSize = size;
    if (list.Count <= maxSize)
    {
        innerCollection = new Collection<T>(list);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}

protected void ThrowTooManyItemsException()
{
    throw new IndexOutOfRangeException(
        "Unable to add any more items, maximum size is " + maxSize.ToString()
        + " items.");
}

#region IList<T> Members

public int IndexOf(T item)
{
    return (innerCollection as IList<T>).IndexOf(item);
}

public void Insert(int index, T item)
{
    if (Count < maxSize)
    {
        (innerCollection as IList<T>).Insert(index, item);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}

public void RemoveAt(int index)
{
    (innerCollection as IList<T>).RemoveAt(index);
}

public T this[int index]
{
    get
    {
        return (innerCollection as IList<T>)[index];
    }
    set
    {
        (innerCollection as IList<T>)[index] = value;
    }
}
```



```
#endregion

#region ICollection<T> Members

public void Add(T item)
{
    if (Count < maxSize)
    {
        (innerCollection as ICollection<T>).Add(item);
    }
    else
    {
        ThrowTooManyItemsException();
    }
}

public void Clear()
{
    (innerCollection as ICollection<T>).Clear();
}

public bool Contains(T item)
{
    return (innerCollection as ICollection<T>).Contains(item);
}

public void CopyTo(T[] array, int arrayIndex)
{
    (innerCollection as ICollection<T>).CopyTo(array, arrayIndex);
}

public int Count
{
    get
    {
        return (innerCollection as ICollection<T>).Count;
    }
}

public bool IsReadOnly
{
    get
    {
        return (innerCollection as ICollection<T>).IsReadOnly;
    }
}

public bool Remove(T item)
{
    return (innerCollection as ICollection<T>).Remove(item);
}

#endregion
```

```

    #region IEnumerable<T> Members

    public IEnumerator<T> GetEnumerator()
    {
        return (innerCollection as IEnumerable<T>).GetEnumerator();
    }

    #endregion
}

```

## 习题 6

不。类型参数 *T* 定义为协变。但协变参数类型只能用作方法的返回值，不能用作方法变元。否则就会得到如下编译错误(假定使用名称空间 *VarianceDemo*):

```

Invalid variance: The type parameter 'T' must be contravariantly valid on
'VarianceDemo.IMethaneProducer<T>.BelchAt(T)' . 'T' is covariant.

```

## 第 13 章

### 习题 1

```

public void ProcessEvent(object source, EventArgs e)
{
    if (e is MessageArrivedEventArgs)
    {
        Console.WriteLine("Connection.MessageArrived event received.");
        Console.WriteLine("Message: {0}",
            (e as MessageArrivedEventArgs).Message);
    }
    if (e is ElapsedEventArgs)
    {
        Console.WriteLine("Timer.Elapsed event received.");
        Console.WriteLine("SignalTime: {0}",
            (e as ElapsedEventArgs).SignalTime);
    }
}

public void ProcessElapsedEvent(object source, ElapsedEventArgs e)
{
    ProcessEvent(source, e);
}

```

注意需要这个额外的 *ProcessElapsedEvent()* 方法，因为 *ElapsedEventHandler* 委托不能转换为 *EventHandler* 委托。*MessageHandler* 委托不需要进行这个转换，因为它的语法与 *EventHandler* 相同：

```

public delegate void MessageHandler(object source, EventArgs e);

```

### 习题 2

修改 *Player.cs*，如下所示(修改了一个方法，添加了两个新方法——代码中的注释说明了这些变化):

```

public bool HasWon()
{
    // get temporary copy of hand, which may get modified.
    Cards tempHand = (Cards)hand.Clone();

    // find three and four of a kind sets
    bool fourOfAKind = false;
    bool threeOfAKind = false;
    int fourRank = -1;
    int threeRank = -1;

    int cardsOfRank;
    for (int matchRank = 0; matchRank < 13; matchRank++)
    {
        cardsOfRank = 0;
        foreach (Card c in tempHand)
        {
            if (c.rank == (Rank)matchRank)
            {
                cardsOfRank++;
            }
        }
        if (cardsOfRank == 4)
        {
            // mark set of four
            fourRank = matchRank;
            fourOfAKind = true;
        }
        if (cardsOfRank == 3)
        {
            // two threes means no win possible
            // (threeOfAKind will only be true if this code
            // has already executed)
            if (threeOfAKind == true)
            {
                return false;
            }
            // mark set of three
            threeRank = matchRank;
            threeOfAKind = true;
        }
    }

    // check simple win condition
    if (threeOfAKind && fourOfAKind)
    {
        return true;
    }

    // simplify hand if three or four of a kind is found, by removing used cards
    if (fourOfAKind || threeOfAKind)
    {
        for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
        {
            if ((tempHand[cardIndex].rank == (Rank)fourRank)

```

```

        || (tempHand[cardIndex].rank == (Rank)threeRank))
    {
        tempHand.RemoveAt(cardIndex);
    }
}

// at this point the method may have returned, because:
// - a set of four and a set of three has been found, winning.
// - two sets of three have been found, losing.
// if the method hasn't returned then either:
// - no sets have been found, and tempHand contains 7 cards.
// - a set of three has been found, and tempHand contains 4 cards.
// - a set of four has been found, and tempHand contains 3 cards.

// find run of four sets, start by looking for cards of same suit in the same
// way as before
bool fourOfASuit = false;
bool threeOfASuit = false;
int fourSuit = -1;
int threeSuit = -1;

int cardsOfSuit;
for (int matchSuit = 0; matchSuit < 4; matchSuit++)
{
    cardsOfSuit = 0;
    foreach (Card c in tempHand)
    {
        if (c.suit == (Suit)matchSuit)
        {
            cardsOfSuit++;
        }
    }
    if (cardsOfSuit == 7)
    {
        // if all cards are the same suit then two runs
        // are possible, but not definite.
        threeOfASuit = true;
        threeSuit = matchSuit;
        fourOfASuit = true;
        fourSuit = matchSuit;
    }
    if (cardsOfSuit == 4)
    {
        // mark four card suit.
        fourOfASuit = true;
        fourSuit = matchSuit;
    }
    if (cardsOfSuit == 3)
    {
        // mark three card suit.
        threeOfASuit = true;
        threeSuit = matchSuit;
    }
}
}

```

```

if (!(threeOfASuit || fourOfASuit))
{
    // need at least one run possibility to continue.
    return false;
}

if (tempHand.Count == 7)
{
    if (!(threeOfASuit && fourOfASuit))
    {
        // need a three and a four card suit.
        return false;
    }

    // create two temporary sets for checking.
    Cards set1 = new Cards();
    Cards set2 = new Cards();

    // if all 7 cards are the same suit...
    if (threeSuit == fourSuit)
    {
        // get min and max cards
        int maxVal, minVal;
        GetLimits(tempHand, out maxVal, out minVal);
        for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
        {
            if (((int)tempHand[cardIndex].rank < (minVal + 3))
                || ((int)tempHand[cardIndex].rank > (maxVal - 3)))
            {
                // remove all cards in a three card set that
                // starts at minVal or ends at maxVal.
                tempHand.RemoveAt(cardIndex);
            }
        }
        if (tempHand.Count != 1)
        {
            // if more than one card is left then there aren't two runs.
            return false;
        }
        if ((tempHand[0].rank == (Rank)(minVal + 3))
            || (tempHand[0].rank == (Rank)(maxVal - 3)))
        {
            // if spare card can make one of the three card sets into a
            // four card set then there are two sets.
            return true;
        }
        else
        {
            // if spare card doesn't fit then there are two sets of three
            // cards but no set of four cards.
            return false;
        }
    }
}

```

```
// if three card and four card suits are different...
foreach (Card card in tempHand)
{
    // split cards into sets.
    if (card.suit == (Suit)threeSuit)
    {
        set1.Add(card);
    }
    else
    {
        set2.Add(card);
    }
}

// check if sets are sequential.
if (isSequential(set1) && isSequential(set2))
{
    return true;
}
else
{
    return false;
}
}

// if four cards remain (three of a kind found)
if (tempHand.Count == 4)
{
    // if four cards remain then they must be the same suit.
    if (!fourOfASuit)
    {
        return false;
    }
    // won if cards are sequential.
    if (isSequential(tempHand))
    {
        return true;
    }
}

// if three cards remain (four of a kind found)
if (tempHand.Count == 3)
{
    // if three cards remain then they must be the same suit.
    if (!threeOfASuit)
    {
        return false;
    }
    // won if cards are sequential.
    if (isSequential(tempHand))
    {
        return true;
    }
}
```

```

        // return false if two valid sets don't exist.
        return false;
    }

    // utility method to get max and min ranks of cards
    // (same suit assumed)
    private void GetLimits(Cards cards, out int maxVal, out int minVal)
    {
        maxVal = 0;
        minVal = 14;
        foreach (Card card in cards)
        {
            if ((int)card.rank > maxVal)
            {
                maxVal = (int)card.rank;
            }
            if ((int)card.rank < minVal)
            {
                minVal = (int)card.rank;
            }
        }
    }

    // utility method to see if cards are in a run
    // (same suit assumed)
    private bool isSequential(Cards cards)
    {
        int maxVal, minVal;
        GetLimits(cards, out maxVal, out minVal);
        if ((maxVal - minVal) == (cards.Count - 1))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

## 第 14 章

### 习题 1

为了结合使用对象初始化和类，必须包含一个默认的非参数构造函数。可以给这个类添加一个，或者删除已有的非默认构造函数。此后，就可以使用如下代码，在一步中实例化和初始化这个类：

```

Giraffe myPetGiraffe = new Giraffe
{
    NeckLength = "3.14",
    Name = "Gerald"
};

```



## 习题 2

错误。使用 `var` 关键字声明变量时，该变量仍是强类型化的，编译器会确定变量的类型。

## 习题 3

可以使用已实现的 `Equals()` 方法。注意不能使用 `==` 运算符来执行这个操作，因为 `==` 运算符会比较变量，确定它们是否引用同一个对象。

## 习题 4

扩展方法必须是静态的：

```
public static string ToAcronym(this string inputString)
```

## 习题 5

必须在静态类中包含扩展方法，它可以在包含客户代码的名称空间中访问。为此，可以在同一个名称空间中包含代码，或者导入包含该类的名称空间。

## 习题 6

一种方式如下：

```
public static string ToAcronym(this string inputString)
{
    return inputString.Trim().Split(' ')
        .Aggregate<string, string>("",
            (a, b) => a + (b.Length > 0 ?
                b.ToUpper()[0].ToString() : ""));
}
```

其中使用了三元运算符以防多个空格引发错误。还要注意需要带两个泛型类型参数的 `Aggregate()` 版本，因为需要一个种子值。

## 第 15 章

## 习题 1

Windows 窗体项目中的 `Program.cs` 文件包含应用程序的 `Main()` 方法。默认情况下，这个方法如下所示：

```
[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

## 代码行

```
Application.EnableVisualStyles();
```

控制 Windows 窗体的可见样式。

请注意这行代码在 Windows 2000 上不起作用。

## 习题 2

TabControl 包含 `SelectedIndexChanged` 事件, 在用户移动到另一个选项卡页时, 该事件可用于执行代码。

- (1) 在 Windows 窗体设计器上, 选择 TabControl, 给它添加两个选项卡。
- (2) 把选项卡命名为 Tab Three 和 Tab Four。
- (3) 选择 TabControl, 添加 `SelectedIndexChanged` 事件, 进入代码窗口。
- (4) 输入下述代码:

```
private void tabControl1_SelectedIndexChanged(object sender,
EventArgs e)
{
    string message = "You changed the current tab to '" +
        tabControl1.SelectedTab.Text + "' from '" +
        tabControl1.TabPages[mCurrentTabIndex].Text + "' ";
    mCurrentTabIndex = tabControl1.SelectedIndex;
    MessageBox.Show(message);
}
```

- (5) 在类的顶部添加私有字段 `mCurrentTabIndex`, 如下所示:

```
partial class Form1 : Form
{
    private int mCurrentTabIndex = 0;
```

- (6) 运行应用程序。

默认显示 TabControl 中的第一个选项卡页, 其索引为 0。把私有字段 `mCurrentTabIndex` 设置为 0, 就可以利用这个默认操作。在 `SelectedIndexChanged` 方法中, 建立要显示的消息。为此, 需要使用 `SelectedTab` 属性, 获取刚才选中的选项卡页的 `Text` 属性, 使用 `TabPages` 集合获取字段 `mCurrentTabIndex` 指定的选项卡页的属性。建立了消息后, 就把 `mCurrentTabIndex` 字段改为指向新选中的选项卡页。

## 习题 3

创建一个派生自 `ListViewItem` 的类, 就可以使用它替代“所需要的”`ListViewItem` 类。这意味着, 尽管 `ListView` 本身不知道类的额外信息, 但可以把额外信息直接存储在 `ListView` 显示的项中。

- (1) 创建一个新类 `FQListViewItem`:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
```

```

namespace ListView
{
    class FQListViewItem : ListViewItem
    {
        private string mFullyQualifiedPath;
        public string FullyQualifiedPath
        {
            get { return mFullyQualifiedPath; }
            set { mFullyQualifiedPath = value; }
        }
    }
}

```

(2) 在 Form.cs 文件中找到 ListViewItem 类型，把它改为 FQListViewItem 类型。

(3) 找到对.Tag 的所有引用，将其改为对.FullyQualifiedPath 的引用。在方法 listViewFilesAndFolders\_ItemActivate 中，把第二行中的所选项转换为 FQListViewItem 项，如下所示：

```

string filename =
    ((FQListViewItem)lw.SelectedItems[0]).FullyQualifiedPath;

```

## 第 16 章

### 习题 1

为此，应创建一个新属性和两个事件。首先创建属性(private int maxLength = 32767)：

```

public int MaxLength
{
    get { return maxLength; }
    set
    {
        if (value >= 0 && value <= 32767)
        {
            maxLength = value;
            if (MaxLengthChanged != null)
                MaxLengthChanged(this, new EventArgs());
            textBoxText.MaxLength = value;
        }
    }
}

```

接着创建两个新事件：

```

public event System.EventHandler MaxLengthChanged;
public event System.EventHandler MaxLengthReached;

```

在窗体设计器中，选中文本框，给 TextChanged 事件添加事件处理程序，代码如下：

```

private void txtLabelText_TextChanged(object sender, EventArgs e)
{
    if (textBoxText.Text.Length >= maxLength)
    {
        if (MaxLengthReached != null)

```

```

        MaxLengthReached(this, new EventArgs());
    }
}

```

在普通的文本框中，文本的最大长度是 `System.Int32` 类型的大小，但默认为 32 767 个字符，这一般足以满足要求。在上述第二步的属性中，检查值是否为负，或者是否超过 32 767，如果是，就忽略更改请求。如果值是可接受的，就设置文本框的 `MaxLength` 属性，引发 `MaxLengthChanged` 事件。

`txtLabelText_TextChanged` 事件处理程序检查文本框中的最大字符数是否等于或超过 `maxLength` 中指定的数字，如果是，就引发 `MaxLengthReached` 事件。

## 习题 2

首先选中状态栏上的 3 个字段，把 `Bold` 的值改为 `false` (展开 `Font` 属性，才能进行修改)。把 3 个字段的 `Enabled` 属性都改为 `True`，再双击 `Bold` 字段，输入如下代码：

```

private void toolStripStatusLabelBold_Click(object sender, EventArgs e)
{
    boldToolStripButton.Checked = !boldToolStripButton.Checked;
}

```

双击 `Italic` 字段，输入如下文本：

```

private void toolStripStatusLabelItalic_Click(object sender, EventArgs e)
{
    italicToolStripButton.Checked = !italicToolStripButton.Checked;
}

```

双击 `Underline` 字段，输入如下文本：

```

private void toolStripStatusLabelUnderline_Click(object sender, EventArgs e)
{
    underlineToolStripButton.Checked = !underlineToolStripButton.Checked;
}

```

3 个单击事件处理程序都切换工具栏按钮的 `Checked` 属性。这会引发 `CheckedChanged` 事件。这些事件处理程序都负责完成所有的工作，所以它们都需要修改，使状态文本也发生变化：

```

private void boldToolStripButton_CheckedChanged(object sender, EventArgs e)
{
    Font oldFont, newFont;

    bool checkState = ((ToolStripButton)sender).Checked;
    oldFont = this.richTextBoxText.SelectionFont;

    if (!checkState)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
    else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

    richTextBoxText.SelectionFont = newFont;
    richTextBoxText.Focus();
}

```

```

        boldToolStripMenuItem.CheckedChanged -= new
        EventHandler(boldToolStripMenuItem_CheckedChanged);
        boldToolStripMenuItem.Checked = checkState;
        boldToolStripMenuItem.CheckedChanged += new
        EventHandler(boldToolStripMenuItem_CheckedChanged);
        //StatusBar
        if (!checkState)
            toolStripStatusLabelBold.Font = new Font(toolStripStatusLabelBold.Font,
            toolStripStatusLabelBold.Font.Style & ~FontStyle.Bold);
        else
            toolStripStatusLabelBold.Font = new Font(toolStripStatusLabelBold.Font,
            toolStripStatusLabelBold.Font.Style | FontStyle.Bold);
    }

    private void italicToolStripButton_CheckedChanged(object sender, EventArgs e)
    {
        Font oldFont, newFont;

        bool checkState = ((ToolStripButton)sender).Checked;
        oldFont = this.richTextBoxText.SelectionFont;

        if (!checkState)
            newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
        else
            newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

        richTextBoxText.SelectionFont = newFont;
        richTextBoxText.Focus();

        italicToolStripMenuItem.CheckedChanged -= new
        EventHandler(italicToolStripMenuItem_CheckedChanged);
        italicToolStripMenuItem.Checked = checkState;
        italicToolStripMenuItem.CheckedChanged += new
        EventHandler(italicToolStripMenuItem_CheckedChanged);
        //StatusBar
        if (!checkState)
            toolStripStatusLabelItalic.Font = new
            Font (toolStripStatusLabelItalic. Font,
            toolStripStatusLabelItalic.Font.Style & ~FontStyle.Italic);
        else
            toolStripStatusLabelItalic.Font = new
            Font (toolStripStatusLabelItalic. Font,
            toolStripStatusLabelItalic.Font.Style | FontStyle.Italic);
    }

    private void UnderlineToolStripButton_CheckedChanged(object sender, EventArgs e)
    {
        Font oldFont, newFont;

        bool checkState = ((ToolStripButton)sender).Checked;
        oldFont = this.richTextBoxText.SelectionFont;

        if (!checkState)
            newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
    }

```

```

else
    newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

richTextBoxText.SelectionFont = newFont;
richTextBoxText.Focus();

underlineToolStripMenuItem.CheckedChanged -= new
EventHandler(underlineToolStripMenuItem_CheckedChanged);
underlineToolStripMenuItem.Checked = checkState;
underlineToolStripMenuItem.CheckedChanged += new
EventHandler(underlineToolStripMenuItem_CheckedChanged);

//StatusBar
if (!checkState)
    toolStripStatusLabelUnderline.Font = new
        Font(toolStripStatusLabelUnderline.Font,
            toolStripStatusLabelItalic.Font.Style & ~FontStyle.Underline);
else
    toolStripStatusLabelUnderline.Font = new
        Font(toolStripStatusLabelUnderline.Font,
            toolStripStatusLabelItalic.Font.
                Style | FontStyle.Underline);
}

```

当选中工具栏按钮时,事件处理程序现在把 StatusStrip 面板的字体改成 Bold、Italic 或 Underline。当未选中这些按钮时,面板的字体就是正常字体。

## 第 17 章

### 习题 1

ClickOnce 部署的优点是安装应用程序的用户不需要管理权限。单击一个超链接,就可以自动安装应用程序。另外,还可以配置应用程序的新版本也自动安装。

### 习题 2

应用程序清单描述了应用程序和需要的权限,部署清单描述了部署配置(例如更新策略)。

### 习题 3

如果安装程序需要管理权限,就需要 Windows Installer,而不是 ClickOnce 部署。

### 习题 4

文件系统编辑器、注册编辑器、文件类型编辑器、用户界面编辑器、定制动作编辑器和启动条件编辑器。

## 第 18 章

### 习题 1

LoginView 控件可以添加到母版页上,使这个信息可用于每个内容页面。下面的代码段表示,用户登录时,会在 LoginView 中显示 LoggedInTemplate。LoggedInTemplate 包含一个 Label 和一个 LinkButton。id 为 InfoLabel 的 Label 控件用来显示用户信息。

```
<asp:LoginView ID="LoginView1" runat="server">
  <LoggedInTemplate>
    <asp:Label ID="InfoLabel" runat="server" Text="Hello, User">
    </asp:Label><br />
    <asp:LinkButton ID="LinkButton1" runat="server"
      OnClick="OnLogout">Logout</asp:LinkButton>
  </LoggedInTemplate>
</asp:LoginView>
```

标签在 Page\_Load 事件处理程序的代码隐藏内容中填充。用户名可以通过 Context 属性访问,User.Identity.Name 返回用户名。

```
protected void Page_Load(object sender, EventArgs e)
{
    Control infoLabel = this.LoginView1.FindControl("InfoLabel");
    if (infoLabel != null)
        (infoLabel as Label).Text = "Welcome, " + Context.User.Identity.Name;
}
```

### 习题 2

DropDownlist 的上一个用法是提供一组定义好的项,供用户选择。现在使用连接到 Events 数据库的 SqlDataSource 来替代:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%%$
  ConnectionStrings:BeginVCSharpEventsConnectionString %>"
  SelectCommand="SELECT [Id], [Title], [Date] FROM [Events]
  ORDER BY [Date]">
</asp:SqlDataSource>
```

设置了 DropDownList 控件,使 DataSourceID 引用 SqlDataSource 后,使 DataTextField 引用 SQL 选择语句中的 Title,以显示事件的标题:

```
<asp:DropDownList ID="dropDownListEvents" runat="server"
  DataSourceID="SqlDataSource1" DataTextField="Title"
  DataValueField="Id">
</asp:DropDownList>
```

### 习题 3

通过 File | New Project | ASP.NET Web Application 菜单创建一个项目,就可以创建一个预先建立了许多项的项目。其中有一个母版页 Site.Master,这个母版页使用了 Styles 文件夹中的一个样式表



Site.css。在母版页中，给站点导航使用了一个 Menu 控件。文件 Default.aspx 和 About.aspx 使用了母版页，所以可以导航到这两个文件。

Account 子文件夹中包含几个文件，它们使用身份验证特性，如 Login.aspx、Register.aspx 和 ChangePassword.aspx。

可以将这个项目作为基础，根据需要添加自己的页面和功能。

## 第 19 章

### 习题 1

选择 File | New | Project，再选择 ASP.NET Empty Web Application 模板，创建一个新的 Web 服务，命名为 CinemaReservation。选择 Project | Add New Item ...，再选择 Web Service 模板，添加一个新的 Web 服务，命名为 CinemaReservation.asmx。

### 习题 2

类的代码如下：

```
public class ReserveSeatRequest
{
    public string Name { get; set; }
    public int Row { get; set; }
    public int Seat { get; set; }
}

public class ReserveSeatResponse
{
    public string ReservationName { get; set; }
    public int Row { get; set; }
    public int Seat { get; set; }
}
```

### 习题 3

对于所有座位，应声明一个 reservedSeats 数组，以便记住预订的座位：

```
private const int maxRows = 12;
private const int maxSeats = 16;
private bool[,] reservedSeats = new bool[maxRows, maxSeats];
```

Web 服务方法的实现代码如下。如果所请求的座位是空闲的，就保留这个座位，并从 Web 服务中返回它。如果该座位不是空闲的，就返回下一个空闲的座位。

```
[WebMethod]
public ReserveSeatResponse ReserveSeat(ReserveSeatRequest req)
{
    ReserveSeatResponse resp = new ReserveSeatResponse();
    resp.ReservationName = req.Name;
    object o = HttpContext.Current.Cache["Cinema"];
```

```

        if (o == null)
        {
            // fill seats with data from the database or a file...
            HttpContext.Current.Cache["Cinema"] = reservedSeats;
        }
        else
        {
            reservedSeats = (bool[,])o;
        }
        if (reservedSeats[req.Row, req.Seat] == false)
        {
            reservedSeats[req.Row, req.Seat] = true;
            resp.Row = req.Row;
            resp.Seat = req.Seat;
        }
        else
        {
            int row;
            int seat;
            GetNextFreeSeat(out row, out seat);
            resp.Row = row;
            resp.Seat = seat;
        }
        return resp;
    }
}

```

#### 习题 4

创建一个新的 Windows 应用程序，添加一个对 Web 服务的服务引用。对 Web 服务的调用如下：

```

private void OnRequestSeat(object sender, EventArgs e)
{
    CinemaService.ReserveSeatRequest req =
        new CinemaService.ReserveSeatRequest();
    req.Name = textName.Text;
    req.Seat = int.Parse(textSeat.Text);
    req.Row = int.Parse(textRow.Text);

    CinemaService.CinemaReservationSoapClient ws =
        new CinemaService.CinemaReservationSoapClient();
    CinemaService.ReserveSeatResponse resp =
        ws.ReserveSeat(req);
    MessageBox.Show(String.Format("Reserved seat {0} {1}",
        resp.Row, resp.Seat));
}

```

## 第 20 章

#### 习题 1

复制网站，会复制运行 Web 应用程序所需的所有文件。Visual Studio 2010 有一个对话框可用于双向复制。目标服务器中的较新文件可以在本地复制。如果源代码不应复制到目标 Web 服务器上，

发布时允许创建程序集，再仅把程序集复制到目标 Web 服务器上。

## 习题 2

复制站点前，需要先在目标服务器上创建虚拟目录。使用安装程序可以在安装过程中在 IIS 内部创建虚拟目录。

## 习题 3

选项有发布到文件系统上，使用 FrontPage Server Extensions 发布到服务器上，通过 FTP 发布，以及使用 1-Click 发布功能发布。这主要取决于当前使用的主机选项以及提供程序所提供的功能。无论如何，都必须在服务器上创建虚拟目录。发布到文件系统上时，需要能访问文件系统，如果自己运行 IIS，就应是这种情况。用 FrontPage Server Extensions 发布时，必须将这些扩展安装在服务器上。通过 FTP 发布时，必须将 FTP 服务器安装在服务器上。通过 1-Click 发布时，提供程序必须支持这个新的发布选项。

## 习题 4

首先使用 IIS 管理工具创建一个 Web 应用程序，再使用 Visual Studio 把 Web 服务文件复制到服务器上。

# 第 21 章

## 习题 1

System.IO

## 习题 2

需要随机访问文件时，或者不处理字符串数据时，就使用 FileStream 对象写入文件。

## 习题 3

- Peek(): 获取文件中下一个字符的值，但不前移到下一个文件位置上
- Read(): 获取文件中下一个字符的值，并前移到下一个文件位置上
- Read(char[] buffer, int index, int count): 从 buffer[index] 开始，把 count 个字符读入 buffer
- ReadLine(): 获取一行文本
- ReadToEnd(): 获取文件中的所有文本

## 习题 4

DeflateStream

## 习题 5

确保它不拥有 Serializable 特性。

## 习题 6

- Changed: 修改文件时发生
- Created: 创建文件时发生
- Deleted: 删除文件时发生
- Renamed: 重命名文件时发生

## 习题 7

添加一个按钮, 切换 `FileSystemWatcher.EnableRaisingEvents` 属性的值。

# 第 22 章

## 习题 1

- (1) 双击 Create Node 按钮, 让事件处理程序执行操作。
- (2) 在创建 XmlComment 后, 插入如下 3 行代码:

```
XmlAttribute newPages = document.CreateAttribute("pages");  
newPages.Value = "1000";  
newBook.Attributes.Append(newPages);
```

## 习题 2

- (1) `//elements`——返回文档中的所有节点。
- (2) `element`——返回文档中的每个元素节点, 但不返回元素根节点。
- (3) `element[@Type='Noble Gas']`——返回其 Type 特性的值是 Noble Gas 的每个元素。
- (4) `//mass`——返回名为 mass 的所有节点。
- (5) `//mass/..`——.. 使 XPath 从选中的节点向上移动一个位置, 这表示, 这个查询返回包含 mass 节点的所有节点。
- (6) `element/specification[mass='20.1797']`——选择包含 mass 节点值为 20.1797 的 specification 元素。
- (7) `element/name[text()='Neon']`——要选择包含测试内容的节点, 可以使用 `text()` 函数, 它会选择文本为 Neon 的 name 节点。

## 习题 3

XML 可以是有效的、格式良好的, 也可以是无效的。只要选择 XML 文档的一部分, 就会得到整个 XML 文档的一部分。这意味着, 所选择的 XML 很可能是无效的。大多数 XML 查看器会拒绝显示格式不正确的 XML, 所以不能在标准 XML 查看器中直接显示许多查询的结果。

## 第 23 章

### 习题 1

```
static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

    var queryResults =
        from n in names
        where n.StartsWith("S")
        orderby n descending
        select n;

    Console.WriteLine("Names beginning with S:");

    foreach (var item in queryResults) {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

### 习题 2

在小于 5 000 000 的集中，查找小于 1000 的数字：

```
static void Main(string[] args)
{
    int[] arraySizes = { 100, 1000, 10000, 100000,
        1000000, 5000000, 10000000, 50000000 };

    foreach (int i in arraySizes) {
        int[] numbers = generateLotsOfNumbers(i);
        var queryResults = from n in numbers
            where n < 1000
            select n;
        Console.WriteLine("number array size = {0}: Count(n < 1000) = {1}",
            numbers.Length, queryResults.Count());
    }

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

### 习题 3

对于  $n < 1000$ ，性能受到的影响并不明显。

```
static void Main(string[] args)
```

```

{
    int[] numbers = generateLotsOfNumbers(12345678);

    var queryResults =
        from n in numbers
        where n < 1000
        orderby n
        select n
        ;

    Console.WriteLine("Numbers less than 1000:");
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}

```

#### 习题 4

对于非常大的子集, 例如  $n > 1000$ , 而不是  $n < 1000$ , 会非常慢:

```

static void Main(string[] args)
{
    int[] numbers = generateLotsOfNumbers(12345678);

    var queryResults =
        from n in numbers
        where n > 1000
        select n
        ;

    Console.WriteLine("Numbers less than 1000:");
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}

```

#### 习题 5

会输出所有名字, 因为没有查询。

```

static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

    var queryResults = names;
}

```

```

        foreach (var item in queryResults) {
            Console.WriteLine(item);
        }

        Console.Write("Program finished, press Enter/Return to continue:");
        Console.ReadLine();
    }
}

```

## 习题 6

```

var queryResults =
    from c in customers
    where c.Country == "USA"
    select c
;
Console.WriteLine("Customers in USA:");
foreach (Customer c in queryResults)
{
    Console.WriteLine(c);
}

```

## 习题 7

```

static void Main(string[] args)
{
    string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
        "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
    // only Min() and Max() are available (if no lambda is used)
    // for a result set like this consisting only of strings
    Console.WriteLine("Min(names) = " + names.Min());
    Console.WriteLine("Max(names) = " + names.Max());
    var queryResults =
        from n in names
        where n.StartsWith("S")
        select n;
    Console.WriteLine("Query result: names starting with S");
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }

    Console.WriteLine("Min(queryResults) = " + queryResults.Min());
    Console.WriteLine("Max(queryResults) = " + queryResults.Max());

    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}

```



## 第 24 章

### 习题 1

使用下面的代码:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;

namespace BegVCSharp_24_exercise1
{
    class Program
    {
        static void Main(string[] args)
        {
            XDocument xdoc = new XDocument(
                new XElement("employees",
                    new XElement("employee",
                        new XAttribute("ID", "1001"),
                        new XAttribute("FirstName", "Fred"),
                        new XAttribute("LastName", "Lancelot"),
                        new XElement("Skills",
                            new XElement("Language", "C#"),
                            new XElement("Math", "Calculus")
                        )
                    ),
                    new XElement("employee",
                        new XAttribute("ID", "2002"),
                        new XAttribute("FirstName", "Jerry"),
                        new XAttribute("LastName", "Garcia"),
                        new XElement("Skills",
                            new XElement("Language", "French"),
                            new XElement("Math", "Business")
                        )
                    )
                );
            Console.WriteLine(xdoc);

            Console.Write("Program finished, press Enter/Return to continue:");
            Console.ReadLine();
        }
    }
}
```

### 习题 2

使用如下代码:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;

namespace BegVCSharp_24_exercises
{
    class Program
    {
        static void Main(string[] args)
        {
            string xmlFileName =
                @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
            XDocument customers = XDocument.Load(xmlFileName);

            Console.WriteLine("Oldest customers: Companies with orders in 1996:");
            var queryResults =
                from c in customers.Descendants("customer")
                where c.Descendants("order").Attributes("orderYear")
                    .Any(a => a.Value == "1996")
                select c.Attribute("Company");

            foreach (var item in queryResults)
            {
                Console.WriteLine(item);
            }
            Console.Write("Press Enter/Return to continue:");
            Console.ReadLine();
        }
    }
}

```

### 习题 3

代码如下:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;

namespace BegVCSharp_24_exercises
{
    class Program
    {
        static void Main(string[] args)
        {
            string xmlFileName =
                @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
            XDocument customers = XDocument.Load(xmlFileName);

            Console.WriteLine(

```

```

        "Companies with individual orders totaling over $10,000");
var queryResults =
    from c in customers.Descendants("order")
    where Convert.ToDecimal(c.Attribute("orderTotal").Value) > 10000
    select new { OrderID = c.Attribute("orderID"),
                Company = c.Parent.Attribute("Company") };

foreach (var item in queryResults)
{
    Console.WriteLine(item);
}
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
    }
}
}

```

#### 习题 4

使用如下代码:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;

namespace BegVCSharp_24_exercises
{
    class Program
    {
        static void Main(string[] args)
        {
            string xmlFileName =
                @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
            XDocument customers = XDocument.Load(xmlFileName);

            Console.WriteLine("Lifetime highest-selling customers: "+
                "Companies with all orders totaling over $100,000");
            var queryResult =
                from c in customers.Descendants("customer")
                where c.Descendants("order").Attributes("orderTotal")
                    .Sum(o => Convert.ToDecimal(o.Value)) > 100000
                select c.Attribute("Company");

            foreach (var item in queryResult)
            {
                Console.WriteLine(item);
            }
            Console.Write("Press Enter/Return to continue:");
            Console.ReadLine();
        }
    }
}

```

## 习题 5

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace BegVCSharp_24_exercise1
{
    class Program
    {
        static void Main(string[] args)
        {
            NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            Console.WriteLine("Product Details");
            var queryResults = from p in northWindEntities.Products
                               select new
                               {
                                   ID = p.ProductID,
                                   Name = p.ProductName,
                                   Price = p.UnitPrice,
                                   Discontinued = p.Discontinued
                               };
            foreach (var item in queryResults)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("Employee Details");
            var queryResults2 = from e in northWindDataContext.Employees
                                select new
                                {
                                    ID = e.EmployeeID,
                                    Name = e.FirstName+" "+e.LastName,
                                    Title = e.Title
                                };
            foreach (var item in queryResults2)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("Press Enter/Return to continue...");
            Console.ReadLine();
        }
    }
}

```

## 习题 6

使用如下代码:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

```

```

namespace BegVCSharp_24_exercise6
{
    class Program
    {
        static void Main(string[] args)
        {
            NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            Console.WriteLine("Top-Selling Products (Sales over $50,000)");
            var queryResults =
                from p in northWindEntities.Products
                where p.Order_Details.Sum(od => od.Quantity * od.UnitPrice) > 50000
                orderby p.Order_Details.Sum(od => od.Quantity * od.UnitPrice) descending
                select new
                {
                    ID = p.ProductID,
                    Name = p.ProductName,
                    TotalSales = p.Order_Details.Sum(od => od.Quantity * od.UnitPrice)
                };
            foreach (var item in queryResults)
            {
                Console.WriteLine(item);
            }

            Console.WriteLine("Press Enter/Return to continue...");
            Console.ReadLine();
        }
    }
}

```

## 习题 7

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace BegVCSharp_24_exercise7
{
    class Program
    {
        static void Main(string[] args)
        {
            NORTHWNDEntities northWindEntities = new NORTHWNDEntities();

            var totalResults = from od in northWindEntities.Order_Details
                               from c in northWindEntities.Customers
                               where c.CustomerID == od.Order.CustomerID
                               select new
                               {
                                   Product = od.Product.ProductName,
                                   Country = c.Country,

```

```

        Sales = od.UnitPrice * od.Quantity
    };

    var groupResults =
        from c in totalResults
        group c by new { Product = c.Product, Country = c.Country } into cg
        select new {
            Product = cg.Key.Product,
            Country = cg.Key.Country,
            TotalSales = cg.Sum(c => c.Sales)
        }
    ;

    var orderedResults =
        from cg in groupResults
        orderby cg.Country, cg.TotalSales descending
        select cg
    ;

    foreach (var item in orderedResults)
    {
        Console.WriteLine("{0,-12}{1,-20}{2,12}",
            item.Country, item.Product, item.TotalSales.ToString("C2"));
    }
    Console.WriteLine("Press Enter/Return to continue...");
    Console.ReadLine();
}
}
}

```

## 第 25 章

### 习题 1

错误。大多数代码都相同，但有一些小区别，例如在 WPF 浏览器应用程序中必须使用 Page 控件，在 WPF 桌面应用程序中必须使用 Window 控件。

### 习题 2

应使用关联属性。在 XAML 中，使用特性语法和<ParentClassName>. <AttributeName>格式的完全限定的特性名来表示关联属性。下面的代码是一个示例；

```

<Tree>
  <Branch Tree.LeafCount="3" />
  <Branch Tree.LeafCount="42" />
</Tree>

```

### 习题 3

语句 b)和 e)是正确的。语句 a)错误，因为.NET 属性是可选的。c)错误，因为一个类可以包含的依赖属性个数是没有限制的。d)错误，因为这是一个最适合的命名约定，而不是强制的。

## 习题 4

应使用 StackPanel 控件。

## 习题 5

命名约定指定，通道事件的名称与所关联的冒泡事件使用的名称相同，但加上前缀 Preview。

## 习题 6

严格地说，这是一个技巧问题，因为可以连续改变任何属性类型。但如果要连续改变的属性类型不是 double、Color 或 Point，就必须创建自己的时间线类，所以一般最好使用这些类型。

## 习题 7

使用动态资源引用可以在运行期间改变资源引用，或者在运行之前不知道该引用是什么时，使用动态资源引用。

# 第 26 章

## 习题 1

上述应用程序都可以。

## 习题 2

数据合同，需要 DataContractAttribute 和 DataMemberAttribute 特性。

## 习题 3

使用.svc 扩展。

## 习题 4

这是一种方式，但把所有 WCF 配置放在一个独立配置文件中通常更简单，例如 web.config 或 app.config。

## 习题 5

```
[ServiceContract]
public interface IMusicPlayer
{
    [OperationContract(IsOneWay=true)]
    void Play();

    [OperationContract(IsOneWay=true)]
    void Stop();

    [OperationContract]
    TrackInformation GetCurrentTrackInformation();
}
```



还需要一个数据合同来封装跟踪信息，在上面的代码中就是 `TrackInformation`。

## 第 27 章

### 习题 1

复合活动包含两部分——活动本身和设计器(XAML)文件，设计器文件定义了活动在屏幕上的布局。复合活动一般派生于 `NativeActivity` 类，且包含一个子活动集合。例如，`Sequence` 活动的 `Activities` 属性就是一个子活动集合。

需要重写 `Execute()` 方法，以便安排子活动运行——可能会随机选择活动，或者同时运行所有活动。给 `Execute()` 方法传送 `NativeActivityContext` 类的一个实例，它可用于安排子活动的执行。

最后一步是创建一个设计器，以允许用户把活动拖放到您的活动上。这里应使用 XAML 定义活动的外观，通常最好参考内置的活动，看看它们是如何实现的，以重用一些 XAML 资源。如果下载 `Reflector` (<http://reflector.red-gate.com>)，就可以使用 BAML 查看器插件(联机搜索它)，对内置程序集使用的资源进行反编译，以便查看用于定义定制复合活动的 XAML。

### 习题 2

`Workflow 4` 有几个活动可以把工作流用作 WCF 服务。最简单的方式是在新建项目对话框的 WCF 项中选择 `WCF Workflow Service Application` 项目类型，接着就可以添加活动，处理入站的方法调用，根据需把结果返回给调用者。

### 习题 3

工作流存在“持久保存”服务的概念，持久保存服务可以保存和重新加载工作流实例。工作流空闲时(即工作流等待某种形式的外部输入或延迟)，就可以进行保存。如果安装了持久保存提供程序，工作流就会保存到该提供程序中。可以使用 SQL Server 提供程序(参见 `System.Activities.Durable-Instancing` 程序集中的 `SqlWorkflowInstanceStore` 类)。如果在 `WorkflowInvoker` 下运行工作流，就不能保存工作流——必须使用 `WorkflowApplication` 或 `WorkflowServiceHost` 类保存工作流。

